Non-Euclidean machine learning for geomechanics

WaiChing Sun

Columbia University, New York, USA

This book chapter is intended to provide a concise review on how to train, verify and validate constitutive models enhanced by graph-theoretic data. We begin our discussion by reviewing basic concepts of the various types of graphs and under what situations does a graph may serve as an ideal data structures for representing data. We then outline a few usages on how to use graph convolutional neural networks to perform embedding and how these embedding can be potentially useful for constitutive modeling, material designs, and inverse problems.

1 Motivations

In 1736, Mathematician Leonhard Euler was asked to solve one of the most famous mathematics problems – Is it possible to visit all the seven bridges at Königsberg, Prussia, on both sides of the Preger River without repeating any path. This is the Seven Bridges of Königsberg problem (see Fig. 1). The issue Euler faced was that there has not yet been techniques invented to formulate the problem in proper abstract terms. Euler solved this problem by inventing these abstract terms, which laid the foundations of modern graph theory. His conclusion is that such a route, which is now referred to as the Eulerian path, does not exist in the Seven Bridges of Königsberg problem.

The abstract terms Euler invented the vertex(or node) set $\mathbb{V}$, which are all the stops a pedestrian may take, and the edge set, which are the seven bridges $\mathbb{E}$. Together, they form a 2-tuple, which is nowadays referred as an undirected graph, or simply a graph, which is often denoted as $\mathbb{G}(\mathbb{V}, \mathbb{E})$. Here, the graph Euler invented represents the topology of the seven bridges, as whether the Eulerian path exists depends on how the bridges are connected. On the other hand, information such as how long the bridges span, and whether the bridges are safe are irrelevant to the existence of the Eulerian path.

Now let's think of a different case. Consider the polycrystal structure, shown in Fig.

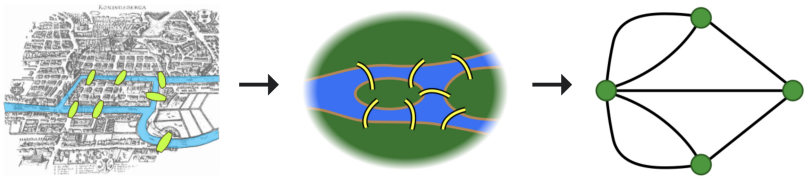


Figure 1: The Seven Bridges of Königsberg problem that leads to the birth of graph theory. Figure obtained from Wikipedia

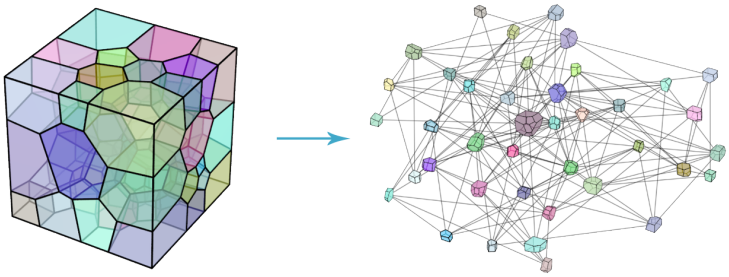


Figure 2: Polycrystal interpreted as a connectivity graph. The graph is undirected and weighted at the nodes. Figure reproduced from Vlassis et al. (2020).

2. Let's say our goal is to determine the effective Young's modulus of this polycrystal. We are given only (1) the orientation of each crystal in this assemble as well as a database which contains the effective Young's modulii of many other polycrystals formed by the same type of crystals.

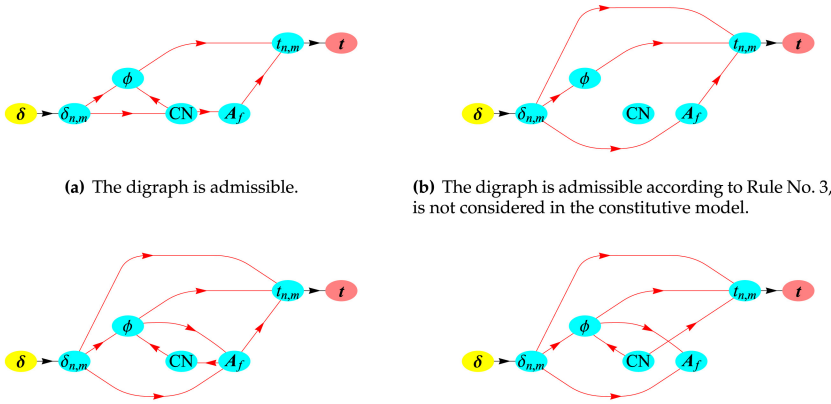
What should we do?

In this case, we involve in a different type of problem in which we are no longer concerning finding a sequence from the edge set $\mathbb{E}$, but we are interested in predictions where we take the micro-structure represented by a graph as the input and output the Young's modulus, i.e.,

$$f : \mathbb{G} \to \mathbb{R}^{+}. \tag{1}$$

The mapping defined in Eq. (1) may look like a function that maps a vector to a real number. However, unlike a vector, where we can get a sense of proximity between two vectors via the inner product or norm equipped by the vector space, it is not trivial to measure the distance between two graphs properly. This issue makes it difficult to perform any interpolation or extrapolation.

In the history of classical mechanics, our intellectual ancestors often bypass this issue by introducing descriptors that implicitly describe some aspect of the microstructures. For instance, void ratio is one of the most used measures of microstructures in soil plasticity model (Schofield and Wroth, 1968; Mitchell et al., 2005; Zhu et al., 2016; Borja, 2013). With this descriptor, we then hypothesize the existence of a critical state as a function of void ratio and stress. Other material models, such as the bounding surface model with critical state that evolves with fabric tensors (Dafalias and Manzari, 2004), breakage theory (Einav, 2007) can be considered as other instances of this descriptor-driven modeling approach. The central idea is to explain the physics by building models that can corroborate well with experimental observations by supplementing a set of hypotheses formulated around these descriptors. Remarkably, the hierarchy of these hypotheses can also be abstracted as a graph, i.e., a directed graph, in which the direction of the edge is defined to represent a hierarchical relation such as cause-and-effort, and orders of a sequence, as shown in Fig. 3.



(a) The digraph is admissible.

(b) The digraph is admissible according to Rule No. 3, CN is not considered in the constitutive model.

(c) The digraph violates Rule No. 4, since there exists a cycle $\phi \to A_f \to CN \to \phi$.

(d) The digraph violates Rule No. 5, since CN and A_f are not on any paths leading from δ to t.

Figure 3: Directed graph used to represent different models of traction-separation law generated by artificial intelligence. Figure reproduced from Wang and Sun (2019).

However, an important issue central to this formalism is that the set of descriptors incorporated into the constitutive laws could be insufficient to accurately and precisely describe the microstructures. For example, if void ratio are the only physical quantities incorporated into the hardening law of a plasticitiy model, then the two specimens with completely different fabrics but with the same void ratio will be considered identical. In other words, the capacity of a model to make accurate, robust and precise predictions depends strongly on the expressivity of the set of descriptors.

1.1 Why machine learning?

Consider again the polycrystal structure shown in Fig. 2. What if we can formulate a method in which we can put all the different graphs that represent the microstructures in a vector space such that the microstructures that are close are given position vectors that are close to each other and microstructures that are distant are given position vectors that are far apart? If this can be achieved, we can leverage the properties of the N-dimensional Euclidean space $\mathbb{R}^N$ to perform regression to formulate material models (Vlassis et al., 2020). This technique is often referred as graph embedding in the literature (Hamilton et al., 2017).

In Vlassis et al. (2020), for instance, a modeling framework is formulated to combine unsupervised learning, which embeds high-dimensional data onto simpler geometry with sufficient smoothness, and supervised mapping, which introduces mapping for labeled data (see Fig. 4) which involves Sobolev training, a technique used for training hyperelasticity material models with gradient data (Vlassis and Sun, 2021; Vlassis et al., 2022). This setup can be considered as a standard pipeline for building models not only in mechanics but in almost every applications including but not limited to computational chemistry, biology, recommendation system and games (Chami, 2021) (see Fig. 4).

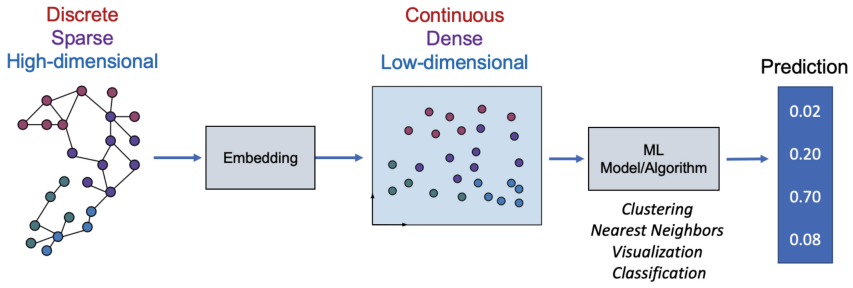


Figure 4: Overview of a typical machine learning pipeline. Figure reproduced from Chami (2021).

1.2 Organization of this Chapter

This chapter provides a brief review on the state-of-the-art of graph embedding and autoencoder techniques used for computational mechanics. We will begin by reviewing the basic terminology for graphs in Section 2. Then, we explain the concepts of graph autoencoder and graph embedding in Section 3 and provide an example in which internal varialbes are generated from the graph neural network (Section 4).

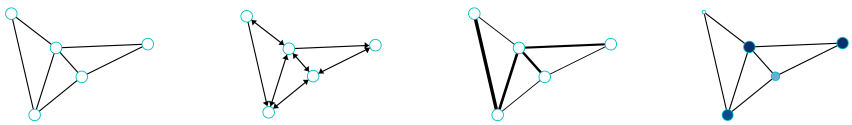
2 Basic terminology of graphs

In this section, a brief review of several terms of graph theory is provided to facilitate the illustration of the concepts in this current work. More elaborate descriptions can be found in Graham et al. (1989); West et al. (2001); Bang-Jensen and Gutin (2008):

Definition 2.1. A **graph** is a two-tuple $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ where $\mathbb{V} = \{v_1, ..., v_N\}$ is a non-empty **vertex set** (also referred to as nodes) and $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ is an **edge set**. To define a graph, there exists a relation that associates each edge with two vertices (not necessarily distinct). These two vertices are called the edge's **endpoints**. The pair of endpoints can either be unordered or ordered.

Definition 2.2. An **undirected graph** is a graph whose edge set $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ connects *unordered* pairs of vertices together.

Definition 2.3. A **loop** is an edge whose endpoint vertices are the same. When all the nodes in the graph are in a loop with themselves, the graph is referred to as allowing self-loops.



(a) (b) (c) (d)

Figure 5: Different types of graphs. (a) Undirected (simple) binary graph (b) Directed binary graph (c) Edge-weighted undirected graph (d) Node-weighted undirected graph.

Definition 2.4. **Multiple edges** are edges having the same pair of endpoint vertices.

Definition 2.5. A **simple graph** is a graph that does not have loops or multiple edges.

Definition 2.6. Two vertices that are connected by an edge are referred to as **adjacent** or as **neighbors**.

Definition 2.7. The term **weighted graph** traditionally refers to graph that consists of edges that associate with edge-weight function $w_{ij} : \mathbb{E} \to \mathbb{R}^n$ with $(i, j) \in \mathbb{E}$ that maps all edges in $\mathbb{E}$ onto a set of real numbers. n is the total number of edge weights and each set of edge weights can be represented by a matrix $\boldsymbol{W}$ with components w_{ij}.

In this current work, unless otherwise stated, we will be referring to weighted graphs as graphs weighted at the vertices - each node carries information as a set of weights

that quantify features of microstructures. All vertices are associated with a vertex-weight function $f_v : \mathbb{V} \to \mathbb{R}^D$ with $v \in \mathbb{V}$ that maps all vertices in $\mathbb{V}$ onto a set of real numbers, where D is the number of weights - features. The node weights can be represented by a $N \times D$ matrix $\boldsymbol{X}$ with components x_{ik}, where the index $i \in [1, ..., N]$ represents the node and the index $k \in [1, ..., D]$ represents the type of node weight - feature.

Definition 2.8. A graph whose edges are unweighted ($w_\epsilon = 1 \ \ \forall \epsilon \in \mathbb{E}$) can be called a **binary graph**.

To facilitate the description of graph structures, several terms for representing graphs are introduced:

Definition 2.9. The **adjacency matrix** $\boldsymbol{A}$ of a graph $\mathbb{G}$ is the $N \times N$ matrix in which entry α_{ij} is the number of edges in $\mathbb{G}$ with endpoints $\{v_i, v_j\}$, as shown in Eq. 2.

$$\alpha_{ij} = \begin{cases} 1, & v_i \text{ is adjacent to } v_j \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

Definition 2.10. If the vertex v is an endpoint of edge ϵ, then v and ϵ are **incident**. The **degree** d of a vertex v is the number of incident edges. The **degree matrix** $\boldsymbol{D}$ of a graph $\mathbb{G}$ is the $N \times N$ diagonal matrix with diagonal entries d_i equal to the degree of vertex v_i, as shown in Eq. (3).

$$deg_{ij} = \begin{cases} d_i, & i = j \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

Definition 2.11. An **isomorphism** from a graph $\mathbb{G}$ to another graph $\mathbb{H}$ is a bijection g that maps $\mathbb{V}(\mathbb{G})$ to $\mathbb{V}(\mathbb{H})$ and $\mathbb{E}(\mathbb{G})$ to $\mathbb{E}(\mathbb{H})$ such that each edge of $\mathbb{G}$ with endpoints u and v is mapped to an edge with endpoints $g(u)$ and $g(v)$. Applying the same permutation to both the rows and the columns of the adjacency matrix of graph $\mathbb{G}$ results to the adjacency matrix of an isomorphic graph $\mathbb{H}$.

Definition 2.12. The **unnormalized Laplacian operator** $\boldsymbol{\Delta}$ is defined such that:

$$(\boldsymbol{\Delta} f)_i = \sum_{j:(i,j)\in\mathbb{E}} w_{ij}(f_i - f_j) \tag{4}$$

$$= f_i \sum_{j:(i,j)\in\mathbb{E}} w_{ij} - \sum_{j:(i,j)\in\mathbb{E}} w_{ij} f_j. \tag{5}$$

By writing the equation above in matrix form, the unnormalized Laplacian matrix $\boldsymbol{\Delta}$ of a graph $\mathbb{G}$ is the $N \times N$ positive semi-definite matrix defined as $\boldsymbol{\Delta} = \boldsymbol{D} - \boldsymbol{W}$.

In this current work, binary graphs will be used, thus, the equivalent expression is used for the unnormalized Laplacian matrix $\boldsymbol{L}$, defined as $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}$ with the entries l_{ij} calculated as:

$$l_{ij} = \begin{cases} d_i, & i = j \\ -1, & i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

Definition 2.13. For binary graphs, the **symmetric normalized Laplacian matrix** $\boldsymbol{L}^{\text{sym}}$ of a graph $\mathbb{G}$ is the $N \times N$ matrix defined as:

$$\boldsymbol{L}^{\text{sym}} = \boldsymbol{D}^{-\frac{1}{2}} \boldsymbol{L} \boldsymbol{D}^{-\frac{1}{2}} = \boldsymbol{I} - \boldsymbol{D}^{-\frac{1}{2}} \boldsymbol{A} \boldsymbol{D}^{-\frac{1}{2}}. \tag{7}$$

The entries l_{ij}^{sym} of the matrix $\boldsymbol{L}^{\text{sym}}$ are shown in Eq. 8.

$$l_{ij}^{\text{sym}} = \begin{cases} 1, & i = j \text{ and } d_i \neq 0 \\ -(d_i d_j)^{-\frac{1}{2}}, & i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

3 Graph embedding

Graph embedding is a technique in which one attempts to create a mapping that maps a graph onto a geometry, which in most case is an Euclidean space. This mapping must be constructed in a way such that the distance between any two graphs are preserved to prevent information loss. In fact, one may consider convolutional neural network also as a special case of graph embedding where the grid is regarded as a graph. This special case is simpler in the sense that a node at any point in the grid system has a fixed number of neighbors (2 for one-dimensional cases, 4 for 2D cases,...etc).

However, there are many systems in mechanics, such as granular assembles, knowledge represented in graphs, where the topology is not fixed. In such a case, the topology of the graphs must be considered in the embedding process. Earlier attempts on graph embedding often relies on the eigen-decomposition of the graph Laplacian to obtain vector representations of graphs. However, the major drawback of this approach is that the the eigen-decomposition only makes sense to compare graphs of the same number of nodes. The eigen-decomposition also makes it difficult to focus on specific local effects in the graphs, which can be quite important for applications, such as fracture or strain localization (Goyal and Ferrara, 2018).

An alternative approach, which we will describe in this section, is to introduce a two-step procedure where one first perform node embedding, i.e., mapping each node of the graphs to a vector in the embedding space (see Fig. 6), then introduce a graph pooling to aggregate information of the entire graph into a single encoded feature

vector. This technique is commonly referred as message-passing graph neural network in the literature.

Earlier work of message-passing graph embedding does not necessarily involve any neural network. For instance, in DeepWalk, Perozzi et al. (2014) uses truncated random walk to aggregate relations among nodes in a graph and use a similarity metric to embed data from social networks. Later work, such as graph convolution neural network (Kipf and Welling, 2016), graph attention neural network (Velickovic et al., 2017), and graph isomorphism network (Xu et al., 2018), all introduce neural networks to aggregate information for the node embedding. A key result for this change in design is to improve the expressivity of the embedding. In particular, we want to be able to distinguish graphs that are different by avoiding them to be mapped onto the same encoded feature vectors. As we will see in the latter discussion, the successful embedding with an expressive neural network is the necessary (but not sufficient) condition for the successful downstream tasks, including the building the decoder (i.e. the inverse mapping from the embedding vector space back to the graph), as well as other prediction or classification problems.

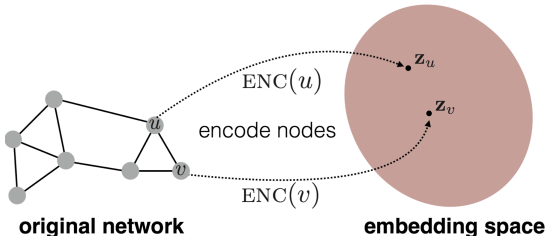


Figure 6: Graph autoencoder architecture. Notice that the input and output graphs are not necessarily sharing the same adjacency matrix.

The rest of this section is organized as follows. We first provide an example of representing finite element solution as a weighted graph (Section 3.1). We then introduce the neural network architecture used in Section 3.2. Then, we present the formulation of these learning tasks: Section 3.3 shows how we learn the mapping for building ${}^{h}\tilde{\mathbb{G}}$ and then find the reduced ordered latent space; Section 3.4 shows how to predict the finite element solutions with $\boldsymbol{p}$ based on the latent space.

3.1 Finite element solution represented by graphs

We consider each node of the finite elements as a graph vertex. Graph edges are assigned for each pair of vertices that are nodes of a finite element edge (Vlassis and Sun, 2023). Accordingly, each nodal solution can be stored as the weight of the corresponding vertex such that the finite element solution can be stored as an undirected node-weighted graph. By assuming that we only use the same set of bases for the testing and interpolating functions of all finite elements, we eliminate the need to in-

troduce edge weights for the edge set. This setting simplifies the graph representation of the finite element solutions.

In the following part of this section, we will introduce the mathematical expression of finite element node graph as a foundation for our machine learning model. A finite element node graph is an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i |\, i = 1, ..., N\}$ is the node set of the graph as v_i corresponding to individual finite element nodes, and $\mathcal{E} = \{(v_{1j}, v_{2j}) |\, j = 1, ..., M;\ \ v_{1j}, v_{2j} \in \mathcal{V}\}$ is the edge set where the existence of each individual edge indicates that node v_{1j} and v_{2j} belong to the same element. N and M indicate the size of the node set and edge set. In order to deal with geometrical features to be incorporated in the machine learning model, we enrich the graph representation as a node-weighted graph: $\mathcal{G}' = (\mathcal{V}, \mathcal{E}, \mathcal{X})$ where $\mathcal{X} = \{\boldsymbol{x}_i \in \mathbb{R}^D | i = 1, ..., N\}$ is the nodal feature set as $\boldsymbol{x}_i$ indicates the geometrical feature vector at node v_i with a dimension of D. We may enforce $D \geq 3$ as the manifold reconstruction task requires at least the three spatial coordinates of the finite element point cloud.

3.2 Graph autoencoder architecture

We adopt the graph isomorphism network (GIN) (cf. Xu et al. (2018)) to perform the embedding task. We would like to discuss GIN, because it is a message-passing model capable of discriminating different graph structures identified by the Weisfeiler-Lehma isomorphism test (Weisfeiler and Leman, 1968). Providing that we use a proper graph pooling layer, the embedding of GIN is inherently permutation invariance, which means that the ordering of the nodes will not affect the predictions. More importantly, the fact that GIN passes the isomorphism test enables us to distinguish non-isomorphic subgraphs by mapping them onto different encoded latent vectors and vice versa – a feat that cannot be achieved by the conventional graph convolutional network and GraphSAGE (Xu et al., 2018). These two features combined improve the expressive power of the GIN such that the relationships among finite element nodes can be captured by the neural network. Fig. 7 shows the architecture of the graph autoencoder designed for the finite element problems. The encoder part of this architecture takes in the adjacency matrix and feature matrix of the input graph $\mathcal{G}'_{\text{in}} : (\boldsymbol{A}_{\text{in}}, \boldsymbol{X}_{\text{in}})$ and produces an encoded feature vector $\boldsymbol{h}_{\text{enc}}$, which is denoted as a functional expression: $\boldsymbol{h}_{\text{enc}} = \text{Enc}(\boldsymbol{X}_{\text{in}}, \boldsymbol{A}_{\text{in}})$. The decoder part of this architecture then utilizes the encoder output to produce a decoded feature matrix $\tilde{\boldsymbol{X}}$. The adjacency matrix of the output graph $\boldsymbol{A}_{\text{out}}$ can be assumed as a prior (since each dataset shares one same $\boldsymbol{A}$) in order to complete an output graph $\mathcal{G}'_{\text{out}} : (\boldsymbol{A}_{\text{out}}, \tilde{\boldsymbol{X}})$, which could be written as $\tilde{\boldsymbol{X}} = \text{Dec}(\boldsymbol{h}_{\text{enc}})$. Problem formulation is generally based on supervision of the decoded output $\tilde{\boldsymbol{X}}$, which will be the focus of the following sections. The rest of this section will present details about how the layer components shown in Fig. 7 operate.

We first introduce one of the most widely-used architectures called multi-layer perceptron (MLP), which is included as a substructure in our graph autoencoder architecture.

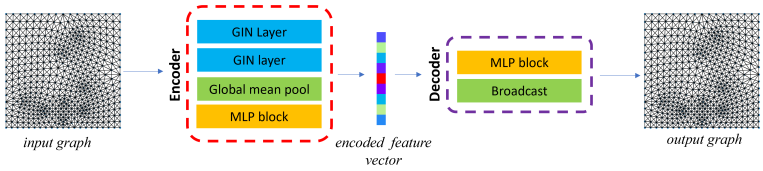


Figure 7: Graph autoencoder architecture. Notice that the input and output graphs are not necessarily sharing the same adjacency matrix.

MLP is a functional approximation expressed as follows:

$$\mathrm{MLP}(\boldsymbol{X}) = \boldsymbol{W}^{(K)} \cdot \mathrm{act}(\boldsymbol{W}^{(K-1)} \cdot \mathrm{act}(...\mathrm{act}(\boldsymbol{W}^{(1)} \cdot \boldsymbol{X} + \boldsymbol{b}^{(1)})...) + \boldsymbol{b}^{(K-1)}) + \boldsymbol{b}^{(K)} \tag{9}$$

where $\boldsymbol{W}$ and $\boldsymbol{b}$ are called the weight matrix and the bias vector of the MLP substructure, respectively. The superscript (K) indicates the K-th layer of the MLP substructure. $\mathrm{act}(\cdot)$ is called the activation function of individual layers; here we adopt the rectified linear unit (ReLU) for $\mathrm{act}(\cdot)$ such that $\mathrm{ReLU}(x) = x$ if $x > 0$ otherwise $\mathrm{ReLU}(x) = 0$.

We then focus on the GIN convolution layers, which take in some adjacency matrix and feature matrix and output an embedded feature matrix. The matrix formulation of a GIN layer is:

$$\boldsymbol{H}^{(k)} = \mathrm{MLP}^{(k)}\left((\boldsymbol{A} + (1+\epsilon)\boldsymbol{I}) \cdot \boldsymbol{H}^{(k-1)}\right) \tag{10}$$

where the superscript (k) indicates the k-th layer in the architecture; $\boldsymbol{H}$ is the embedded nodal feature matrix coming from the output of the previous layer with $\boldsymbol{H}^{(1)} = \boldsymbol{X}$ at the input layer. ϵ is a learnable parameter. For consecutive GIN convolution layers, the following layer accepts the same $\boldsymbol{A}$ as the previous layer. For the beginning GIN layer in both the encoder and the decoder, $\boldsymbol{A}$ should be prescribed as either $\boldsymbol{A}_{\mathrm{in}}$ or $\boldsymbol{A}_{\mathrm{out}}$.

Our architecture also includes global operations on the graph. The graph global mean pooling operation in the encoder performs the following computation:

$$\boldsymbol{h}_{\mathrm{avg}} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{h}_i \tag{11}$$

where $\boldsymbol{h}_i$ is the embedded nodal feature of v_i corresponding to the i-th row of $\boldsymbol{H}$, and $\boldsymbol{h}_{\mathrm{avg}}$ is the resultant graph feature vector from global mean pooling.

The broadcasting operation in the decoder is a matrix reshape operation that converts a row vector $\boldsymbol{h}_{\mathrm{dec}}$ of size ND_{enc} coming from the output of an MLP substructure in the decoder, to an embedded nodal feature matrix of size $N \times D_{\mathrm{enc}}$ such that:

$$h_{ij} = (\boldsymbol{h}_{\mathrm{dec}})_{j+(i-1)D_{\mathrm{enc}}} \tag{12}$$

where h_{ij} indicates the j-th component of the embedded nodal feature vector $\boldsymbol{h}_i$ after broadcasting.

3.3 Learning problem for the finite element or discrete element simulations

In the previous section, we discuss the strategy to construct the mapping $\mathcal{F}({}^{l}\mathcal{G}'_j) = {}^{h}\tilde{\mathcal{G}}'_j$ such that ${}^{h}\tilde{\mathcal{G}}'_j \sim {}^{l}\mathcal{G}'_j$ to establish an augmented data set ${}^{h}\tilde{\mathbb{G}}$. We assume that $\mathcal{F}$ is learned in a supervised manner: we aim to minimize the discrepancy between the nodal feature matrix of training labels and the approximated nodal feature matrix output by the neural network. We intuitively construct the training labels with ${}^{h}\mathbb{G}$, and thus we required that for each snapshot in ${}^{h}\mathbb{G}$ there exist some snapshot in ${}^{l}\mathbb{G}$ corresponding to the results with the same loading condition, which is summarized as follows:

$$\exists\, {}^{lh}\mathbb{G} \subset {}^{l}\mathbb{G} \quad \text{s.t.} \quad {}^{h}\mathbb{G} = \left\{ \mathcal{F}({}^{lh}\mathcal{G}'_j) \middle| \, {}^{lh}\mathcal{G}'_j \in {}^{lh}\mathbb{G} \right\} \tag{13}$$

The subset ${}^{lh}\mathbb{G}$ then constructs the training input set. We next approximate $\mathcal{F}$ with the graph autoencoder proposed in Section 3.2 mainly computing the decoded nodal features as $\tilde{\boldsymbol{X}} = \hat{\boldsymbol{F}}(\boldsymbol{X})$. The loss function is adopted as the node-wise mean square error of the nodal feature matrix, which leads to the following training objective:

$$\min_{\boldsymbol{\Theta}^F} \frac{1}{N_s^{lh}} \sum_{i=1}^{N_s^{lh}} \left\| \hat{\boldsymbol{F}}(\boldsymbol{X}^{lh}_{(i)}) - \boldsymbol{X}^{h}_{(i)} \right\|^2_{\text{fro}}, \quad \hat{\boldsymbol{F}}(\boldsymbol{X}) = \text{Dec}_F(\text{Enc}_F(\boldsymbol{X}, \boldsymbol{A}^l)) \tag{14}$$

where $\boldsymbol{\Theta}^F$ is the collection of all trainable network parameters of $\hat{\boldsymbol{F}}(\cdot)$. N_s^{lh} is the size of ${}^{lh}\mathbb{G}$ as well as ${}^{h}\mathbb{G}$. The subscript (i) indicates the i-th sample in ${}^{h}\mathbb{G}$ or ${}^{lh}\mathbb{G}$, while the sample sequence satisfies ${}^{h}\mathcal{G}'_i \sim {}^{lh}\mathcal{G}'_i$ with ${}^{h}\mathcal{G}'_i : (\boldsymbol{A}^h, \boldsymbol{X}^h_{(i)})$ and ${}^{lh}\mathcal{G}'_i : (\boldsymbol{A}^l, \boldsymbol{X}^{lh}_{(i)})$. The operator $\|\cdot\|_{\text{fro}}$ indicates the Frobenius norm of a matrix. The subscript F of Dec and Enc indicates the decoder and encoder function for the graph autoencoder approximating $\mathcal{F}$, in order to differentiate from the reconstruction autoencoder mentioned in the following part. The approximated mapping $\hat{\boldsymbol{F}}(\cdot)$ helps us to enrich the high-fidelity dataset as follows:

$${}^{h}\tilde{\mathbb{G}} = \left\{ {}^{h}\tilde{\mathcal{G}}'_j : (\boldsymbol{A}^h, \tilde{\boldsymbol{X}}^h_{(j)}) \mid \tilde{\boldsymbol{X}}^h_{(j)} = \begin{cases} \hat{\boldsymbol{F}}(\boldsymbol{X}^l_{(j)}), & {}^{l}\mathcal{G}'_j : (\boldsymbol{A}^l, \boldsymbol{X}^l_{(j)}) \in {}^{l}\mathbb{G} \setminus {}^{lh}\mathbb{G}, \\ \boldsymbol{X}^l_{(j)}, & {}^{l}\mathcal{G}'_j : (\boldsymbol{A}^l, \boldsymbol{X}^l_{(j)}) \in {}^{lh}\mathbb{G}. \end{cases} \right\} \tag{15}$$

Here we realize the significance of having a low-fidelity dataset ${}^{l}\mathbb{G}$ and the mapping $\mathcal{F}$: we are now able to construct a relatively large high-fidelity dataset to improve the reduced ordered model without spending too much effort performing experiments on the high-fidelity scales.

With ${}^{h}\tilde{\mathbb{G}}$ populated from ${}^{l}\mathbb{G}$, we are ready to formulate the graph embedding problem that determines the reduced ordered latent space. The general idea is to construct

a graph autoencoder function $\hat{\boldsymbol{R}}(\cdot)$ whose output approximates its input itself. We still adopt the loss function as the node-wise mean square error for the feature matrix between the labels in ${}^{h}\tilde{\mathbb{G}}$ and the output from $\hat{\boldsymbol{R}}(\cdot)$, which yields the following training objective:

$$\min_{\boldsymbol{\Theta}^R} \frac{1}{N_s^{hh}} \sum_{i=1}^{N_s^{hh}} \left\| \hat{\boldsymbol{R}}(\tilde{\boldsymbol{X}}_{(i)}^{h}) - \tilde{\boldsymbol{X}}_{(i)}^{h} \right\|_{\text{fro}}^2, \quad \hat{\boldsymbol{R}}(\boldsymbol{X}) = \text{Dec}_R(\text{Enc}_R(\boldsymbol{X}, \boldsymbol{A}^h)) \tag{16}$$

where $\boldsymbol{\Theta}^R$ is the collection of all trainable network parameters of $\hat{\boldsymbol{R}}(\cdot)$. N_s^{hh} is the size of ${}^{h}\tilde{\mathbb{G}}$ as well as ${}^{l}\mathbb{G}$. The subscript R indicates the encoder and decoder function of the reconstruction autoencoder. The reduced ordered latent space $\mathbb{L}$ is then defined as the space spanned by $\boldsymbol{h}_{\text{enc}} = \text{Enc}_R(\boldsymbol{X}, \boldsymbol{A}^h)$ for arbitrary $\boldsymbol{X}$ coming from an admissible deformed configuration $\mathcal{G}'$, where $\mathbb{L} \subset \mathbb{R}^{D_{\text{enc}}}$.

3.4 Predicting the high-fidelity results without full-scale simulations

This section presents how we utilize the parametrized load $\boldsymbol{p}$ to predict the actual deformed configuration $\mathcal{G}'$. As we find the reduced ordered latent space $\mathbb{L}$ in the previous section, we may notice that $\mathbb{L}$ is generally entangled, which does not ideally captures the reduced ordered dynamics in the maximum sense. We here propose to construct a response controlling law on $\mathbb{L}$ based on $\boldsymbol{p}$ corresponding to the configuration of interest to disentangle the reduced ordered latent space, denoted as a functional expression $\boldsymbol{h}_{\text{enc}} = f(\boldsymbol{p})$. We then approximately parametrize the response controlling law function as $\hat{\boldsymbol{f}}(\cdot)$ by a simple feed-forward neural network with MLP architecture. We fit $\hat{\boldsymbol{f}}(\cdot)$ with the mean square error loss between the encoded feature vector obtained from the graph autoencoder and that computed by $\hat{\boldsymbol{f}}(\cdot)$, which yields the following learning objective:

$$\min_{\boldsymbol{\Theta}^f} \frac{1}{N_s^{hh}} \sum_{i=1}^{N_s^{hh}} \left\| \hat{\boldsymbol{f}}(\boldsymbol{p}_{(i)}) - \text{Enc}_R(\tilde{\boldsymbol{X}}_{(i)}^{h}, \boldsymbol{A}^h) \right\|_2^2 \tag{17}$$

where $\boldsymbol{\Theta}^f$ is the collection of all trainable network parameters of $\hat{\boldsymbol{f}}(\cdot)$. $\boldsymbol{p}_{(i)}$ is the loading condition corresponding to the i-th snapshot ${}^{h}\tilde{\mathcal{G}}_i' : (\boldsymbol{A}^h, \tilde{\boldsymbol{X}}_{(i)}^{h})$. The operator $\|\cdot\|_2$ is the vector Euclidean norm.

After we learn the neural network approximation of $f(\cdot)$, the prediction of high-fidelity results is to determine the nodal feature matrix $\bar{\boldsymbol{X}}$ given some loading condition $\bar{\boldsymbol{p}}$ as follows:

$$\bar{\boldsymbol{X}} = \text{Dec}_R(\hat{\boldsymbol{f}}(\bar{\boldsymbol{p}})) \tag{18}$$

In essence, we introduce a graph neural network approach to construct a response surface. Since (1) the data are obtained from the simulations that obey the balance principles and (2) a successful embedding should be capable of preserving the relationships among nodes, we hypothesize that this will give us more accurate and robust

predictions than other alternatives that employ basis functions that directly interpolate the hypersurface in the ambient space $\mathbb{R}^{N+P}$.

4 Numerical Example: Training of interpretable graph embedding internal variables

In this section, we describe a procedure of embedding field simulation data to construct graph-based internal variables. The content of this section is revised from a section in Vlassis and Sun (2023).

A graph convolutional autoencoder is used to compress the graph structures that carry the plastic deformation distribution of a microstructure. In Section 4.1, we demonstrate the process of generating the plasticity data through finite element method (FEM) simulations and post-processing them into weighted graph structures. In Section 4.2, we showcase the performance capacity of the autoencoder architecture as well as its ability to reproduce the plasticity graph structures. Finally, in Section 4.3, we perform a sensitivity training test for the autoencoder architecture on different FEM meshes for the same microstructure.

4.1 Generation of the plasticity graph database

In this work, the autoencoders used for the generation of the graph-based internal variables and the neural network constitutive models used for the forward predictions are trained on data sets generated by FEM elastoplasticity simulations. To test the autoencoders' capacity to generate encoded feature vectors regardless of the microstructure and plastic strain distribution patterns the FEM mesh represents, we test the algorithm with two microstructures of different levels of complexity. The two microstructures A and B are demonstrated in Fig. 8 (a) and (b) respectively. The outline of the microstructures is a square with a side of 1 mm. This figure also shows the meshing of the two microstructures. The microstructures A and B are discretized by 250 and 186 triangular elements respectively with one integration point each. An investigation of different mesh sizes and the sensitivity of the encoded feature generation is demonstrated in Section 4.3. Each integration point of mesh corresponds to a node in the equivalent graph (also shown in Fig. 8). The integration points of the neighboring elements - elements that share at least one vertex - are connected with an edge in the constructed graph.

The constitutive model selected for the local behavior at the material points was linear elasticity and J2 plasticity with isotropic hardening. The local behavior is predicted with an energy minimization algorithm. The local optimization algorithm is omitted for brevity. The local linear elastic material has a Young's modulus of $E = 2.0799$MPa and a Poisson ratio of $\nu = 0.3$. The local J2 plasticity has an initial yield stress of 100kPa and a hardening modulus of $H = 0.1E$. During the simulation, the elastic, plastic, and total strain as well as the hyperelastic energy functional and stress are saved for every integration point.

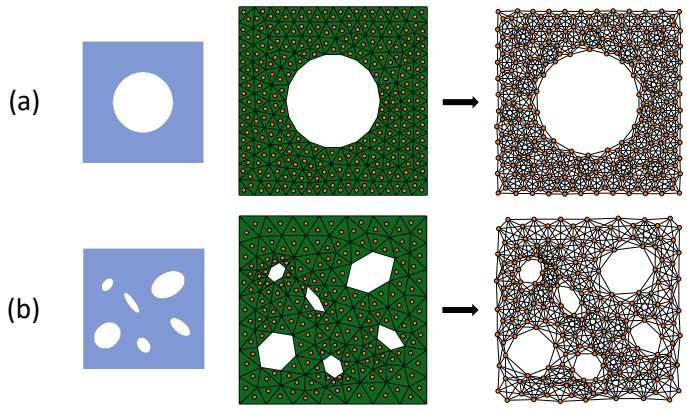


Figure 8: Two microstructures represented as a finite element mesh and an equivalent node-weighted undirected graph.

To capture varying patterns of distribution of plastic strain, the finite element simulations were performed under various combinations of uniaxial and shear loading. The loading was enforced with displacement boundary conditions applied to all the sides of the mesh for both microstructures A and B. The combinations of displacement boundary conditions are sampled by rotating a loading displacement vector from 0° to 90° whose components are the uniaxial displacements in the two directions for the pure axial displacement cases and the uniaxial and shear displacements for the combined uniaxial and shear loading. The maximum displacement magnitude for axial and shear loading vector components are $u_{\text{goal}} = 1.5 \times 10^{-3}$ mm. We sample a total of 100 loading combinations/FEM simulations for each microstructure. During each of these simulations, we record the constitutive response at every material point and post-process it as a node-weighted graph and a volume average response. For every simulation, we record 100 time steps, thus collecting 10000 training sample pairs of graphs and homogenized responses for each microstructure.

4.2 Training of the graph autoencoder

In this section, we demonstrate the training performance of the autoencoder architecture on the two microstructure data sets described in Section 4.1. We also show the capacity of the autoencoder to reproduce the plasticity graphs in the training samples. The dimension of the encoded feature vector in this example is set to $D_{\text{enc}} = 16$. An examination of the effect of the encoded feature vector size is described in Section 4.3.

The training curves for the autoencoder's reconstruction loss function is shown in Fig. 9. The autoencoder appears to have similar loss function performance for both microstructures. The autoencoder performs slightly better for microstructure A as it is tasked to learn and reproduce patterns for a seemingly simpler microstructure

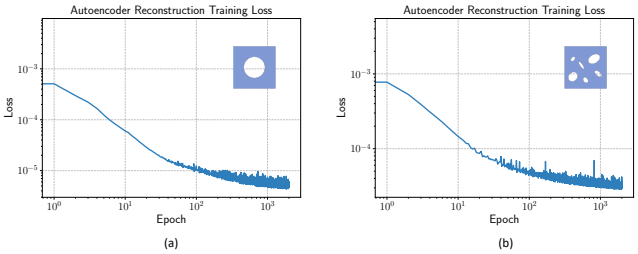


Figure 9: Autoencoder reconstruction training loss for microstructures A and B (a and b respectively).

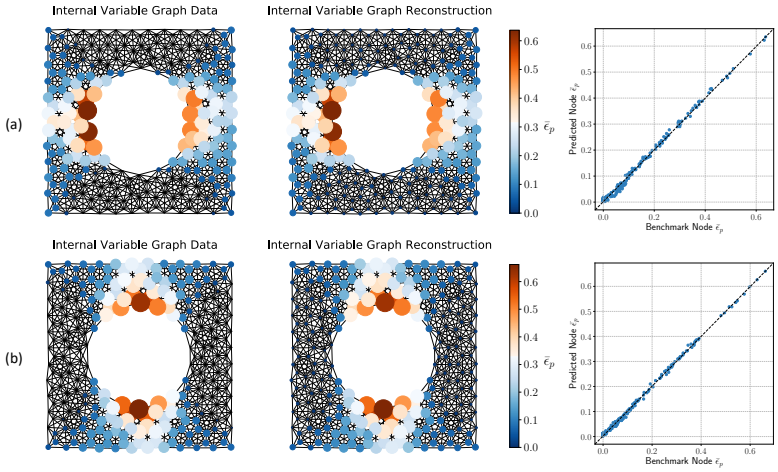


Figure 10: Prediction of the autoencoder architecture of the microstructure A for two loading paths (a and b). The graph node size and color represent the magnitude of the accumulated plastic strain $\bar{\epsilon}_p$. The node-wise predictions for $\bar{\epsilon}_p$ are also demonstrated.

compared to microstructure B. The training loss curves in this figure demonstrate the overall performance of the autoencoder architecture – the encoder and decoder components of the architecture are trained simultaneously. The capacity of the autoencoder to reconstruct the plasticity distribution patterns is explored in Fig. 10 and Fig. 11 for microstructures A and B respectively. In these figures, we showcase the reconstruction capacity of the plastic strain for two different time steps for each microstructure. The time steps selected are from two different loading path combinations resulting in different plasticity graph patterns. We demonstrate how the autoencoder can reproduce these patterns by comparing the internal variable graph data – the autoencoder input – with the graph reconstruction – the autoencoder output. We also show the accuracy of the node-wise prediction of the accumulated plastic strain for these microstructures. It

is noted that the autoencoder predicts the values of the full plastic strain tensor at the nodes. However, these plots show the accumulated plastic strain $\overline{\epsilon}_p$ values calculated from the predicted strain tensor at the nodes for easier visualization.

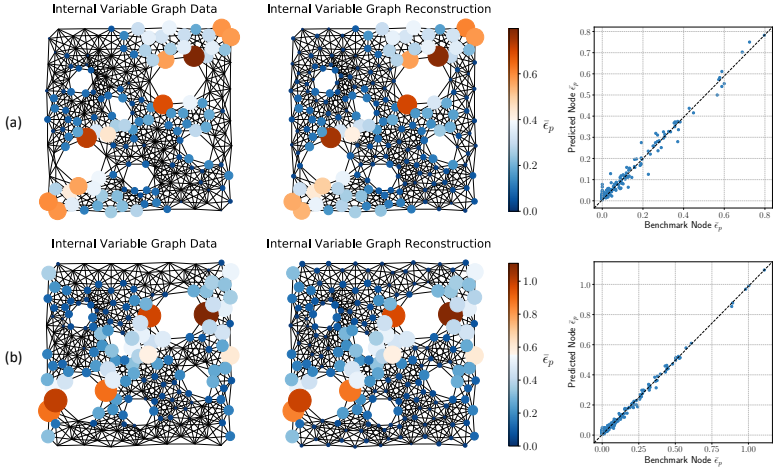


Figure 11: Prediction of the autoencoder architecture of the microstructure B for two loading paths (a and b). The graph node size and color represent the magnitude of the accumulated plastic strain $\overline{\epsilon}_p$. The node-wise predictions for $\overline{\epsilon}_p$ are also demonstrated.

The autoencoder architecture provides the flexibility of utilizing its two components, the encoder $\mathcal{L}_{\text{enc}}$ and the decoder $\mathcal{L}_{\text{dec}}$, separately. In this section, we demonstrate the encoder's ability to process the high-dimensional graph structure in encoded feature vector $\boldsymbol{\zeta}$ time histories. In Fig. 12 (a) & (c) and Fig. 13 (a) & (c), we show the predicted encoded feature vector $\boldsymbol{\zeta}_n$ for a time step of a loading path for microstructures A and B respectively. These encoded feature vectors specifically correspond to the graphs shown in Fig. 10 and Fig. 11 respectively. In Fig. 12 (b) & (d) and Fig. 13 (b) & (d), we demonstrate the time series of plasticity graphs encoded in time series of encoded feature vectors. It is highlighted that the encoded feature vector values do not change during the elastic/path-independent part of the loading. This is directly attributed to the fact that the plasticity graph is constant (zero plastic strain at the nodes) before yielding for all the time steps. The benefit of separately using the decoder $\mathcal{L}_{\text{dec}}$ as a post-processing step to interpret the predicted encoded feature vectors is also explored in the following sections.

4.3 Mesh sensitivity and Encoded Feature Vector dimension

In this section, we investigate the behavior for different dimensionality of the graph data set and the compression of the graph information. In the first experiment, we test the effect of the size of the input plasticity graph that will be reconstructed by the autoencoder. We generate three data sets from finite element simulations with

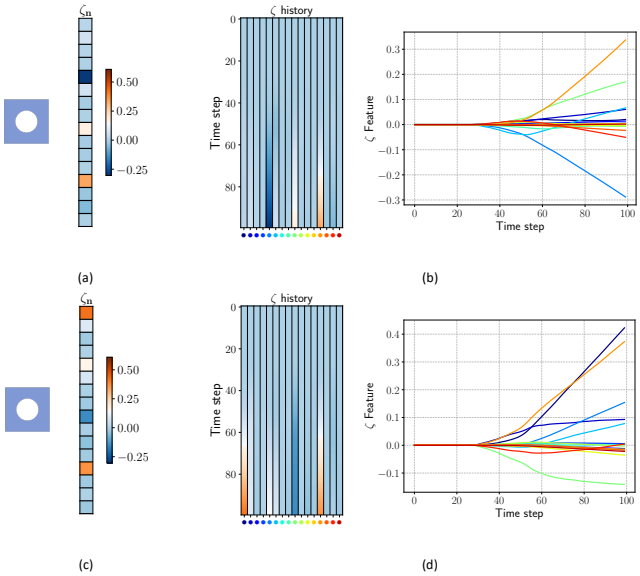


Figure 12: Prediction of encoded feature vector $\boldsymbol{\zeta}$ by the encoder $\mathcal{L}_{\text{enc}}$ for microstructure A. (a,c) The encoded feature vector $\boldsymbol{\zeta}_n$ for a single time step for the plastic graphs shown in Fig. 10a and Fig. 10b respectively. (b,d) The encoded feature vector $\boldsymbol{\zeta}$ history for all the time steps in the loading paths of Fig. 10a and Fig. 10b respectively.

different mesh sizes for microstructure A. All meshes consist of the same triangular elements described in the previous section. The number of elements in the mesh and the corresponding nodes in the post-processed graphs are $N = 100$, $N = 250$, and $N = 576$. For the mesh generation, we start with the $N = 100$ mesh and refine once to obtain the $N = 250$ mesh and twice to obtain the $N = 576$ mesh. The refinement was performed automatically using the meshing software library Cubit (Blacker et al., 1994). The data sets for the meshes of $N = 100$, $N = 250$, and $N = 576$ nodes mesh were generated through the same FEM simulation setup and a subset of the combinations of uniaxial and shear loading paths described in Section 4.1 gathering 2500 training samples of graphs.

The results of the training experiment on the mesh sensitivity are demonstrated in Fig. 14. The figure shows the reconstruction loss for the three mesh sizes. The reconstruction loss exhibits a minor improvement as the number of nodes in the graph increase. This is attributed to the density of the information available for the autoencoder to learn the patterns from – there is a higher resolution of adjacent nodes' features. However, an increase in graph size may increase the duration of the training procedure. Since the benefit of increasing the mesh size is not significant in this set of numerical experiments, we opt for the $N = 250$ node mesh to use for the rest of this work.

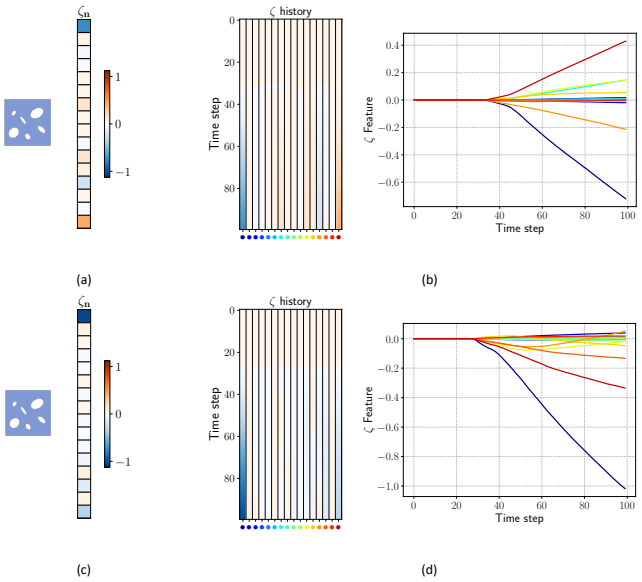


Figure 13: Prediction of encoded feature vector $\boldsymbol{\zeta}$ by the encoder $\mathcal{L}_{\text{enc}}$ for microstructure B. (a,c) The encoded feature vector $\boldsymbol{\zeta}_n$ for a single time step for the plastic graphs shown in Fig. 11a and Fig. 11) respectively. (b,d) The encoded feature vector $\boldsymbol{\zeta}$ history for all the time steps in the loading paths of Fig. 11a and Fig. 11b respectively.

In a second numerical experiment, we examine the effect of the size of the encoded feature vector on the capacity of the autoencoder to learn and reconstruct the plasticity distribution patterns. We re-train the autoencoder using the data set of 10000 graphs generated on the $N = 250$ node mesh for microstructure A as described in Section 4.1. We perform three training experiments selecting different sizes D_{enc} for the encoded feature vector – $D_{\text{enc}} = 2$, $D_{\text{enc}} = 16$, and $D_{\text{enc}} = 32$. All the convolutional filters and the Dense layers have the same size. The only Dense layers that are affected are those around the encoded feature vector whose input and output sizes are modified to accommodate the different sizes of encoded feature vector.

The training performance for these three training experiments is demonstrated in Fig. 15. Compared to encoded feature vector sizes $D_{\text{enc}} = 16$ and $D_{\text{enc}} = 32$, the $D_{\text{enc}} = 2$ autoencoder architecture seems to fail to compress the information as well with a loss performance difference of about two orders of magnitude. The maximum compression achieved for this autoencoder architecture setup appears to be two features. This dimensionality appears to be the smallest feasible encoding limit for this particular data set. It is also possible that more sampling from different loading paths may also increase this minimal dimensionality. Jumping from $D_{\text{enc}} = 16$ to $D_{\text{enc}} = 32$ encoded feature vector components, only a small improvement in the reconstruction capacity is observed. The reconstruction capacity is also illustrated in Fig. 16. The

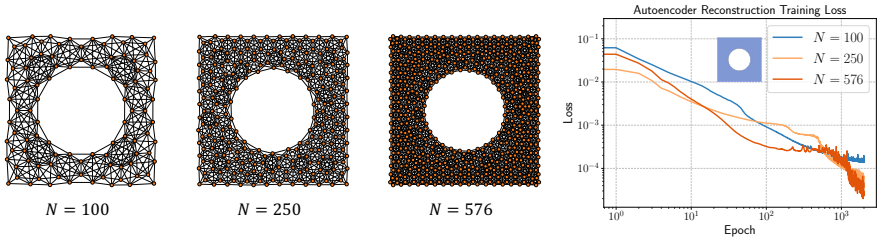


Figure 14: Autoencoder reconstruction training loss for microstructure A with the size of the encoded feature vector $D_{\text{enc}} = 16$ and graph sizes of $N = 100$, $N = 250$, and $N = 576$ nodes.

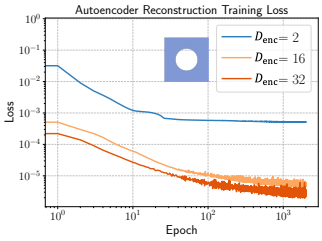


Figure 15: Autoencoder reconstruction training loss for microstructure A with the size of the encoded feature vector $D_{\text{enc}} = 2$, $D_{\text{enc}} = 16$, and $D_{\text{enc}} = 32$.

decoder fails to accurately reconstruct the plasticity graph from the $D_{\text{enc}} = 2$ encoded feature vector (Fig. 16 a). However, for $D_{\text{enc}} = 16$ and $D_{\text{enc}} = 32$ the decoder accurately reproduces the plasticity patterns (Fig. 16 b and c). It is expected for dimensions larger than $D_{\text{enc}} = 32$ the benefit in reconstruction capacity will me minimal. Thus, the encoded feature vector dimension selected for the rest of this work is $D_{\text{enc}} = 16$ for which the dimension reduction capacity is considered adequate and computationally efficient.

It should be noted that the relatively small losses observed in Fig. 14 only indicate that the three reconstructions are independently successful in the sense that the reconstructed graphs are all sufficiently close to the original ones.

4.4 Return mapping algorithm

In this section, we provide the implementation details for the return mapping algorithm to make forward elasoplasticity predictions. The fully implicit stress integration algorithm allows for the incorporation of the graph-based internal variables generated from the autoencoder architecture in the prediction scheme. The return mapping algorithm designed in the principal strain space is described in Algorithm 1.

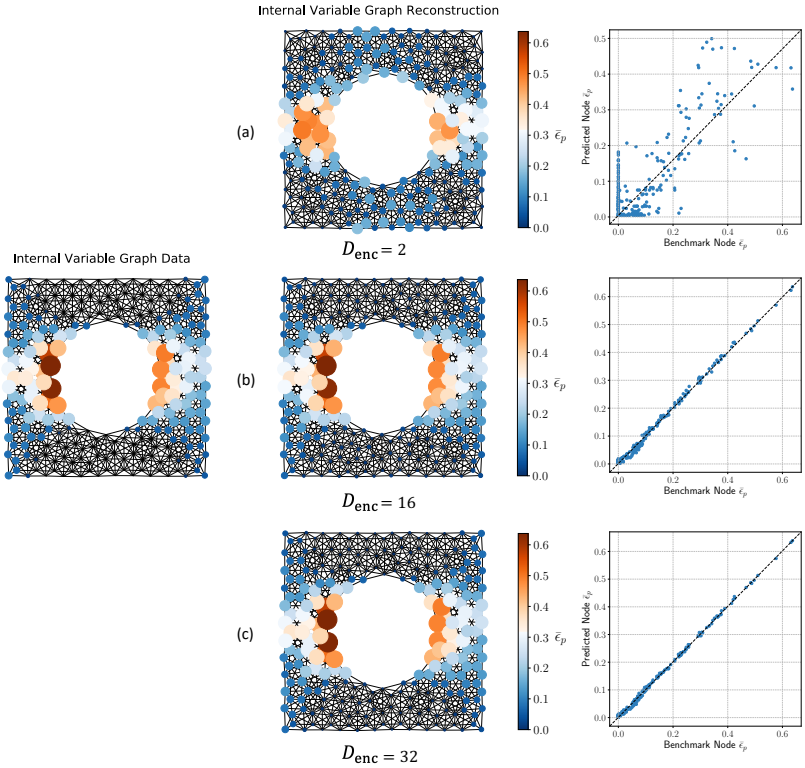


Figure 16: Comparison of the autoencoder architecture's capacity to reconstruct the accumulated plastic strain pattern of the microstructure A with the size of the encoded feature vector (a) $D_{\text{enc}} = 2$, (b) $D_{\text{enc}} = 16$, and (c) $D_{\text{enc}} = 32$. The graph node size and color represent the magnitude of the accumulated plastic strain $\bar{\epsilon}_p$. The node-wise predictions for $\bar{\epsilon}_p$ are also demonstrated.

This formulation of the return mapping algorithm requires all the strain and stress measures are in principal axes. However, this is not limiting for the choice of strain and stress space formulation of the constitutive law components as the framework allows for coordinate system transformation through automatic differentiation. The automatic differentiation is facilitated with the use of the Autograd library (Maclaurin et al., 2015). Through automatic differentiation and a series of chain rules the constitutive model predictions are expressed in the principal axes, allowing for any invariant formulation of the yield function during training.

The elastoplastic behavior is modeled through a predictor-corrector scheme that integrates the elastic prediction with the corrections by the yield function neural network. It is noted that the elastic update predictions for the hyperelastic energy functional and the plasticity terms encountered in the return mapping algorithm are evaluated as

neural network predictions using the offline trained energy functional, yield function, and kinetic law. The hyperelastic neural network is based on the prediction of an energy functional with interpretable first-order and second-order derivatives (stress and stiffness respectively).

Besides the capacity to predict the values of the approximated functions, these libraries also allow for the automatic evaluation of the approximated function derivatives that are required to perform the return mapping constitutive updates and constructing the local Newton-Raphson tangent matrix as well as the necessary coordinate set transformation chain rules.

Algorithm 1 Return mapping algorithm in strain-space for encoded feature vector internal variable plasticity.

Require: Hyperelastic energy functional $\widehat{\psi}^{\mathrm{e}}$ neural network, yield function $\widehat{f}$ neural network, the encoded feature vector neural network $\widehat{\boldsymbol{\zeta}}$, and the plastic flow network $\widehat{\boldsymbol{g}}$.

1. Compute trial elastic strain

Compute $\boldsymbol{\epsilon}^{\mathrm{e}\,\mathrm{tr}}_{n+1} = \boldsymbol{\epsilon}^{\mathrm{e}}_{n} + \Delta\boldsymbol{\epsilon}$.

Spectrally decompose $\boldsymbol{\epsilon}^{\mathrm{e}\,\mathrm{tr}}_{n+1} = \sum_{A=1}^{3} \epsilon^{\mathrm{e}\,\mathrm{tr}}_{A} \boldsymbol{n}^{\mathrm{tr},A} \otimes \boldsymbol{n}^{\mathrm{tr},A}$.

2. Compute trial elastic stress

Compute $\sigma^{\mathrm{tr}}_{A} = \partial\widehat{\psi}^{\mathrm{e}}/\partial\epsilon^{\mathrm{e}}_{A}$ for $A = 1, 2, 3$ and the corresponding $p^{\mathrm{tr}}, q^{\mathrm{tr}}$ at $\epsilon^{\mathrm{e}\,\mathrm{tr}}_{n+1}$.

3. Check yield condition and perform return mapping if loading is plastic

if $\widehat{f}\,(p^{\mathrm{tr}}, q^{\mathrm{tr}}, \xi_n) \leq 0$ **then**

Set $\boldsymbol{\sigma}_{n+1} = \sum_{A=1}^{3} \sigma^{\mathrm{tr}}_{A} \boldsymbol{n}^{\mathrm{tr},A} \otimes \boldsymbol{n}^{\mathrm{tr},A}$ and exit.

else

Compute encoded feature vector $\boldsymbol{\zeta}_n = \widehat{\boldsymbol{\zeta}}(\boldsymbol{\epsilon}^{p}_{\mathrm{hist}n})$.

Compute plastic flow direction $\frac{\partial\widehat{\vartheta}}{\partial\sigma_A} = \widehat{\boldsymbol{g}}(\boldsymbol{\delta\zeta}_n)$ for $A = 1, 2, 3$.

Solve for $\epsilon^{\mathrm{e}}_{1}, \epsilon^{\mathrm{e}}_{2}, \epsilon^{\mathrm{e}}_{3}$, and ξ_{n+1} such that $\widehat{f}\,(p, q, \xi_{n+1}) = 0$.

Compute $\boldsymbol{\sigma}_{n+1} = \sum_{A=1}^{3} \left(\partial\widehat{\psi}^{\mathrm{e}}/\partial\epsilon^{\mathrm{e}}_{A}\right) \boldsymbol{n}^{\mathrm{tr},A} \otimes \boldsymbol{n}^{\mathrm{tr},A}$ and exit.

The return mapping also incorporates a non-associative flow rule to update the plastic flow direction instead of using the stress gradient of the yield function. This is achieved by incorporating the predictions of the plastic flow $\widehat{\boldsymbol{g}}$ network. Through the local iteration scheme, the solution for the true elastic strain values can be retrieved using the solved for discrete plastic multiplier $\Delta\lambda$ and the predicted flow direction as follows:

$$\epsilon^{\mathrm{e}}_{A} = \epsilon^{\mathrm{e}\,\mathrm{tr}}_{A} - \Delta\lambda \frac{\partial\widehat{\vartheta}}{\partial\sigma_A} = \epsilon^{\mathrm{e}\,\mathrm{tr}}_{A} - \Delta\lambda \widehat{g}_A(\widehat{\boldsymbol{\delta\zeta}}), \quad A = 1, 2, 3. \tag{19}$$

The return mapping algorithm requires a hyperelastic energy functional neural network $\widehat{\psi}^{\mathrm{e}}$, a yield function $\widehat{f}$, a kinetic law $\widehat{\boldsymbol{\zeta}}$, and a plastic flow $\widehat{\boldsymbol{g}}$ neural network that are pre-trained offline. Given the elastic strain tensor at the current loading step, a

trial elastic stress state is calculated using the hyperelastic energy functional neural network. The yield condition is checked for the trial elastic stress state and the current plastic strain level. If the predicted yield function is positive, the trial stress is the in the elastic region and is the actual stress. The encoded feature vector remains constant. If the yield function is non-positve the trial stress is in the inadmissible stress region and an Newton-Raphson optimization scheme is utilized to correct the stress prediction. The current encoded feature vector is predicted from the time history of plastic strain tensors and is used to predict the current plastic flow directions. The goal of the return mapping algorithm is to solve for the prinicipal elastic strains and the plastic strain such that the predicted yield function is equal to zero and the stress updates is consistent with the plastic flow. The encoded feature vector at every step can be converted back into the corresponding weighted graph via the graph decoder $\mathcal{L}_{\text{dec}}$ neural network. This weighted graph can be converted back into information in a finite element mesh and therefore enable us to interpret the microstructure.

It is noted that the return mapping algorithm is formulated via the principle direction is provided in this section for the generalization purpose. This setting is sufficient for isotropic materials. In our numerical examples, we only introduce two-dimensional cases to illustrate the ideas for simplicity. The generalization of the return mapping algorithm for anisotropic materials is straightforward, but the training of the yield function and the plastic flow model in the higher dimensional parametric space is not trivial. This improvement will be considered in the future but is out of the scope of this study.

4.5 Interpretable multiscale plasticity of complex microstructures

In this last section, we demonstrate the capacity of the models to make forward predictions for unseen loading paths and interpret them in the microstructure. The return mapping algorithm does not only predict the strain-stress response of the material but also the plastic strain response and the encoded feature vector variables. These can then be decoded by the graph decoder $\mathcal{L}_{\text{dec}}$ to interpret the microstructures' elastoplastic behavior. We provide tests of unseen loading path simulations for both microstructures A and B. The training of the constitutive models used to make the forward predictions is described in Section 4.2.

We first test the models' capacity to make predictions of the plastic state on monotonic data. We demonstrate the result for the predicted stress state in Fig. 17 (a & b) and Fig. 18 (a & b) for microstructures A and B respectively. We also record the homogenized plastic strain tensor of the microstructures. For simplicity, we are demonstrating the predicted accumulated plastic strain measure $\bar{\epsilon}_p$ in Fig. 17 (c) and Fig. 18 (c). Using the trained kinetic law neural network, we can make forward predictions of the encoded feature vectors $\widehat{\boldsymbol{\zeta}}$ that are consistent with the current predicted homogenized plastic state. The results of these predictions are shown in 17 (d) and Fig. 18 (d). These predicted curves are a close match to the benchmark data and can closely capture the behavior in the macroscale. We can now interpret this homogenized behavior as the corresponding one in the microscale. Using the trained decoder for each microstruc-

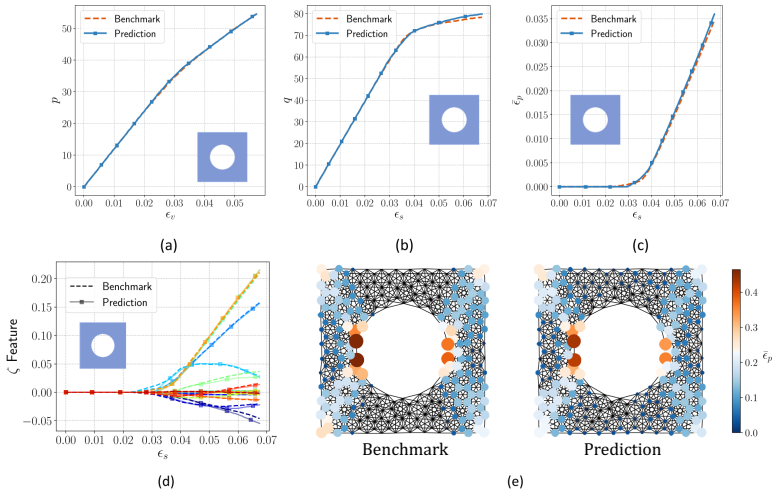


Figure 17: (a,b,c) Prediction of stress invariants p, q, and accumulated plastic strain $\bar{\epsilon}_p$ using the return mapping algorithm for a monotonic loading of microstructure A. (d,e) Prediction of all the encoded feature vector $\boldsymbol{\zeta}$ components and the corresponding decoded internal variable graph for a monotonic loading of microstructure A.

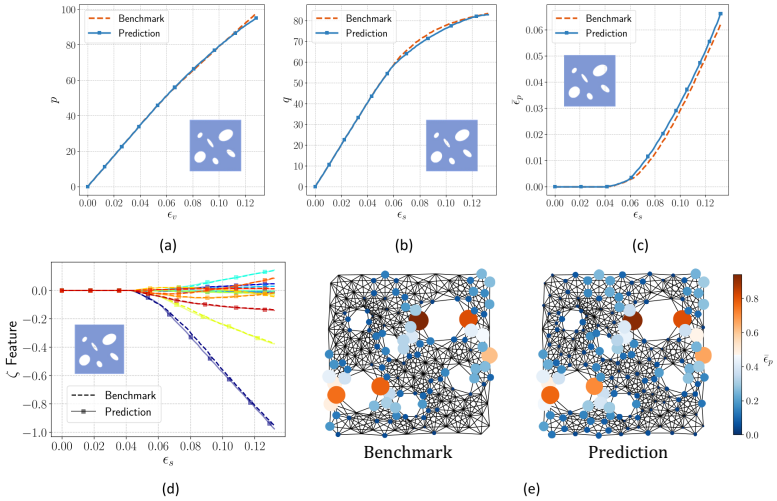


Figure 18: (a,b,c) Prediction of stress invariants p, q, and accumulated plastic strain $\bar{\epsilon}_p$ using the return mapping algorithm for a monotonic loading of microstructure B. (d,e) Prediction of all the encoded feature vector $\boldsymbol{\zeta}$ components and the corresponding decoded internal variable graph for a monotonic loading of microstructure B.

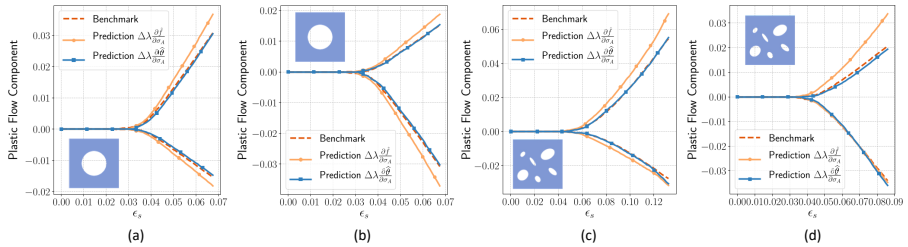


Figure 19: Prediction of the plastic flow components for two loading cases for microstructures A and B (a & b respectively). Both the predicted via yield function stress gradient $\frac{\partial\hat{f}}{\partial\sigma_A}$ and the non-associative plastic flow $\frac{\partial\hat{\vartheta}}{\partial\sigma_A}$ predictions are shown.

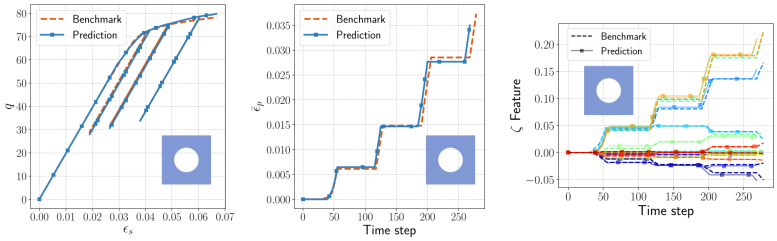


Figure 20: Prediction of deviatoric stress q, accumulated plastic strain $\bar{\epsilon}_p$, and the encoded feature vector $\boldsymbol{\zeta}$ components using the return mapping algorithm for a cyclic loading of microstructure A.

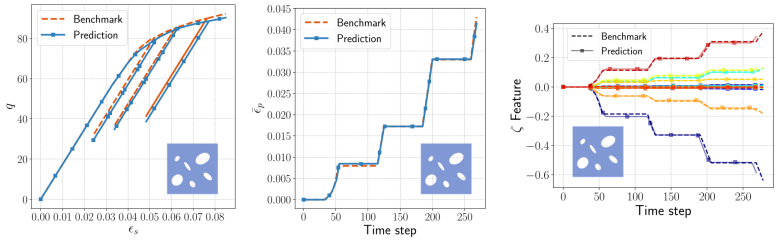


Figure 21: Prediction of deviatoric stress q, accumulated plastic strain $\bar{\epsilon}_p$, and the encoded feature vector $\boldsymbol{\zeta}$ components using the return mapping algorithm for a cyclic loading of microstructure B.

ture, we can recover the plastic strain distributions as shown in Fig.17 (e) and Fig. 18 (e) for microstructures A and B respectively. It is noted that while only the node-wise prediction of the accumulated plastic strain $\bar{\epsilon}_p$ is shown, the decoded recovers the entire plastic strain tensor $\boldsymbol{\epsilon}_p$. This is done for simplicity of presentation. The node-wise

predictions of the plastic strain are accurate and the decoder can qualitatively capture the general plastic distribution patterns and the plastic strain localization nodes in the microstructure.

We also demonstrate the capacity of the model to predict the plastic flow with the help of the encoded feature vector internal variables. In Fig. 19, we compare the computed plastic flow components using the neural network yield function stress gradient $\frac{\partial \widehat{f}}{\partial \sigma_A}$ prediction and the plastic potential stress gradient $\frac{\partial \widehat{\vartheta}}{\partial \sigma_A}$ as predicted by the plastic flow $\widehat{\boldsymbol{g}}$ network with the benchmark simulations. The results demonstrated are for two blind prediction curves for each microstructure – Fig. 19(a) corresponds to Fig. 17 and Fig. 19(c) corresponds to Fig. 18. The accuracy of the $\widehat{\boldsymbol{g}}$ network predictions on the flow is higher than that of the yield function stress gradient. This is attributed to the decoupling of the yielding and hardening from the plastic flow directions allowing for more flexibility of the neural networks to fit these complex laws. Network $\widehat{\boldsymbol{g}}$ also utilizes the highly descriptive encoded feature vector $\widehat{\boldsymbol{\zeta}}$ input that allows for more refined control of the plastic flow than the volume averaged accumulated plastic strain metric used in the yield function formulation.

Finally, we conduct a similar blind test experiment but with added blind unloading and reloading elastic paths in the loading strains. The results for microstructures A and B are demonstrated in Fig. 20 and Fig. 21 respectively. The model does not have any difficulty recognizing the elastic and plastic regions of the loading path since the behaviors are distinguished with the help of the neural network yield function. This also constrains the evolution of the plastic strain and the encoded feature vector to happen only during the plastic loading. Since the kinetic law neural network is a feed-forward architecture, there is no history dependence and no change in the plastic strain corresponds to no change in the encoded feature vector. The decoder architecture is also path-independent so no change in the encoded feature vector corresponds to no change in the respective decoded plastic graph. This is also achieved by the specific way the encoded feature vectors are constructed. The input node features in the autoencoder are the mesh node coordinates and the plastic strains. This specific design ensures that the plastic graph does not evolve during elastic unloading/reloading and prevents any artificial memory effect in the elastic regime. The lack of memory effect in the elastic regime is necessary for the encoded feature vector to be internal variables for rate-independent plasticity models where the history-dependent effect is only triggered once the yield criterion is met. This would not be the case if other integration point data, such as the total strain or stress, are incorporated into the graph encoder.

Note that this switch between path-independent and path-dependent behaviors may also have implications for other neural network constitutive laws. In particular, if a black-box recurrent neural network is used to forecast history-dependent stress-strain responses, then one must ensure that the history-dependent effect is not manifested in the elastic region. For instance, if the LSTM architecture is used, then one must ensure that the forget gate is trained to turn on to filter out any potential artificial influence of the strain history.

5 Conclusion

This book chapter reviews the applications of the message-passing graph neural network for mechanics problems. We provide an example of how we can use graph embedding to create encoder that compress history-dependent spatial pattern into descriptors and state variables for plasticity models.

Acknowledgments.

This book chapter is written while the author is on sabbatical at Stanford University under the support of the UPS Foundation Visiting Professorship. This support is gratefully acknowledged.

References

Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2008.

Ted D Blacker, William J Bohnhoff, and Tony L Edwards. Cubit mesh generation environment. volume 1: Users manual. Technical report, Sandia National Labs., Albuquerque, NM (United States), 1994.

Ronaldo I Borja. *Plasticity*, volume 2. Springer, 2013.

Ines Chami. *Representation Learning and Algorithms in Hyperbolic Spaces*. Stanford University, 2021.

Yannis F Dafalias and Majid T Manzari. Simple plasticity sand model accounting for fabric change effects. *Journal of Engineering mechanics*, 130(6):622–634, 2004.

Itai Einav. Breakage mechanics—part i: theory. *Journal of the Mechanics and Physics of Solids*, 55(6):1274–1297, 2007.

Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

Ronald L Graham, Donald E Knuth, Oren Patashnik, and Stanley Liu. Concrete mathematics: a foundation for computer science. *Computers in Physics*, 3(5):106–107, 1989.

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML workshop*, volume 238, 2015.

James Kenneth Mitchell, Kenichi Soga, et al. *Fundamentals of soil behavior*, volume 3. John Wiley & Sons New York, 2005.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

Andrew Noel Schofield and Peter Wroth. *Critical state soil mechanics*, volume 310. McGraw-hill London, 1968.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. Graph attention networks. *stat*, 1050(20):10–48550, 2017.

Nikolaos N Vlassis and WaiChing Sun. Sobolev training of thermodynamic-informed neural networks for interpretable elasto-plasticity models with level set hardening. *Computer Methods in Applied Mechanics and Engineering*, 377:113695, 2021.

Nikolaos N Vlassis and WaiChing Sun. Geometric learning for computational mechanics part ii: Graph embedding for interpretable multiscale plasticity. *Computer Methods in Applied Mechanics and Engineering*, 404:115768, 2023.

Nikolaos N Vlassis, Ran Ma, and WaiChing Sun. Geometric deep learning for computational mechanics part i: anisotropic hyperelasticity. *Computer Methods in Applied Mechanics and Engineering*, 371:113299, 2020.

Nikolaos N Vlassis, Puhan Zhao, Ran Ma, Tommy Sewell, and WaiChing Sun. Molecular dynamics inferred transfer learning models for finite-strain hyperelasticity of monoclinic crystals: Sobolev training and validations against physical constraints. *International Journal for Numerical Methods in Engineering*, 123(17):3922–3949, 2022.

Kun Wang and WaiChing Sun. Meta-modeling game for deriving theory-consistent, microstructure-based traction–separation laws via deep reinforcement learning. *Computer Methods in Applied Mechanics and Engineering*, 346:216–241, 2019.

Boris Weisfeiler and Andrei Leman. The reduction of a graph to canonical form and the algebra which appears therein. *NTI, Series*, 2(9):12–16, 1968.

Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

Huaxiang Zhu, Hien NG Nguyen, François Nicot, and Félix Darve. On a common critical state in localized and diffuse failure modes. *Journal of the Mechanics and Physics of Solids*, 95:112–131, 2016.