Learning to fail: Predicting fracture evolution in brittle material models using recurrent graph convolutional neural networks

Max Schwarzer[a,b], Bryce Rogan[a,b], Yadong Ruan[a], Zhengming Song[a], Diana Y. Lee[a,c], Allon G. Percus[a,*], Viet T. Chau[d], Bryan A. Moore[e], Esteban Rougier[d], Hari S. Viswanathan[d], Gowri Srinivasan[f]

[a] *Institute of Mathematical Sciences, Claremont Graduate University, Claremont, CA 91711, USA*
[b] *Mathematics Department, Pomona College, Claremont, CA 91711, USA*
[c] *Computational Science Research Center, San Diego State University, San Diego, CA 92182, USA*
[d] *Earth and Environmental Sciences Division, Los Alamos National Laboratory, Los Alamos, NM 87545, USA*
[e] *Broadridge Financial Solutions, New York, NY 10016, USA*
[f] *X Computational Physics Division, Los Alamos National Laboratory, Los Alamos, NM 87545, USA*

ARTICLE INFO

ABSTRACT

We propose a machine learning approach to address a key challenge in materials science: predicting how fractures propagate in brittle materials under stress, and how these materials ultimately fail. Our methods use deep learning and train on simulation data from high-fidelity models, emulating the results of these models while avoiding the overwhelming computational demands associated with running a statistically significant sample of simulations. We employ a graph convolutional network that recognizes features of the fracturing material and a recurrent neural network that models the evolution of these features, along with a novel form of data augmentation that compensates for the modest size of our training data. We simultaneously generate predictions for qualitatively distinct material properties. Results on fracture damage and length are within 3% of their simulated values, and results on time to material failure, which is notoriously difficult to predict even with high-fidelity models, are within approximately 15% of simulated values. Once trained, our neural networks generate predictions within seconds, rather than the hours needed to run a single simulation.

1. Introduction

Brittle materials, such as ceramics, glasses, and some metal alloys, are important in a variety of contexts. By definition, these are materials that break with little plastic deformation, meaning that they fracture without bending or stretching to more than a minimal extent. Brittle material failure occurs under stress and can result in catastrophic damage, in turn leading to significant financial and human loss. Historical failures such as the Schenectady Liberty ship and Aloha Airlines Flight 243 demonstrate this clearly: the Schenectady broke nearly in half within only 16 days of service, due to the brittle fracture of low-grade steel components, and the Boeing 737 used for Flight 243 suffered extensive damage due to multisite fatigue cracking of the skin panels, which in turn caused in-flight explosive decompression at 24,000 feet [1].

Microstructural information and the dynamic process of fracture propagation play key roles in understanding the mechanisms of brittle material failure, enabling predictions on whether, when, and how material failure might occur. Applications of fracture dynamics include hydraulic mining in shale gas systems [2], fluid migration [3], and aircraft safety assessment [4]. Numerous methods have been developed to model and study these dynamics. A fruitful approach has been the use of hybrid finite-discrete element methods (FDEM) [5], including the Hybrid Optimization Software Suite (HOSS) [6]. This high-fidelity model simulates fracture and fragmentation processes or materials deformation in both 2D and 3D complex systems, providing accurate predictions of fracture growth leading to material failure. However, running such simulations can require many thousands of hours for materials on the scale of meters, making them computationally prohibitive.

In order to reduce computation time dramatically while retaining accuracy, we propose a machine learning approach to predict dynamic fracture propagation. Our method uses deep learning to model simultaneously a broad range of evolving features in a fracturing

---

material. We convert HOSS simulation results into training data for a neural network that learns the temporal dynamics of a fracture system. Processing the data and training the neural network can both be parallelized efficiently. While this process can take on the order of hours, subsequently using the trained network on an initial set of microcracks can yield predictions within seconds for fracture growth statistics and time to material failure. We thereby speed up computation by many orders of magnitude.

Our approach involves transforming the HOSS FDEM mesh into a fracture graph, where vertices represent fractures and edge weights represent relationships between fractures [7,8]. As features in the graph, we define fracture attributes that include physical features such as fracture length, damage, tip stress, position and orientation, as well as connectivity metrics that include coalescence, damage between fractures, and multiple forms of physical distance. Based on a single graph and its features, our goal is to produce a full time series of graphs that predicts the evolution of these features, mirroring the full dynamics simulated by HOSS. The fracture graph representation also results in a vast reduction of data size, from nearly a terabyte of data for the output of a set of HOSS simulations to only about 20 GB for a set of graph time series.

We use deep learning to predict the evolution of our fracture graphs. Deep learning uses large artificial neural networks for capturing complex nonlinear relationships hidden in data, and has been successful in areas such as automatic speech recognition, image recognition, and natural language processing. Neural networks are well-suited to our problem because a single network can predict the simultaneous evolution of multiple features. Our network architecture exploits the capabilities of a graph convolutional network (gCNs) [9] to recognize features from the reduced graph representation of large fracture networks, coupled with a recurrent neural network (RNN) [10] to model the evolution of those features.

We train our network on fracture graph time series from 145 HOSS simulations. Once trained, the network takes a $t = 0$ graph as input and outputs its prediction for the graph at all subsequent time steps $t = 1, 2, ..., n$. We find that these predictions closely match the actual statistics of HOSS simulations. The predicted fracture size is, on average, within 2% of its simulated value, and the predicted distribution of fracture sizes is consistent with the simulated distribution (at the 0.05 significance level) in 87% of the simulations. For the time to failure, the mean absolute error of predictions is approximately 15% of the average.

In Section 2, we provide the physical and algorithmic background for our approach. Section 3 describes the physical system and the HOSS simulations representing it. Section 4 gives the reduced graph representation we use for our fracture systems, and Section 5 discusses our ML methods. Section 6 presents the results of these methods, and Section 7 summarizes our work and discusses its implications.

2. Background

HOSS [6] is a fracture simulator based on the combined finite-discrete element method (FDEM). FDEM has been used since the 1990s [11–13] to model fragmentation in brittle materials, notably the transition from continuum to discrete behavior that takes place when the material fails. The FDEM formulation describes fracture and fragmentation processes explicitly based on conservation laws, avoiding unnecessary assumptions regarding the behavior of the model.

HOSS simulations have been used to study fracture initiation and propagation in shale rocks in order to enhance the efficiency of hydraulic fracturing [14] and to model earthquake ruptures [15]. While highly accurate, HOSS faces limitations due to computational intensity. Solid domains are discretized, with each fracture represented by tens to hundreds of finite elements. A material on the scale of millimeters can contain 10^6 microcracks, and require 10^7–10^9 elements [16]. Because of the explicit scheme used by HOSS, with Newton's laws providing the

governing equations, correctly resolving the physics of such a material requires very small time steps. Consequently, even simulations involving laboratory-size samples of several centimeters, with thousands of microcracks, can require thousands of hours of computation to model milliseconds of dynamics and create petabytes of data. Additionally, the sample's initial configuration of defects and microcracks is rarely known precisely. This necessitates an uncertainty quantification framework, requiring thousands of simulations to provide statistically significant results on system behavior. Such an approach becomes computationally intractable for materials of practical interest, which are often on the scale of meters.

Machine learning (ML) methods can help mitigate the need for overwhelming computational power in modeling fracture dynamics, by rapidly predicting the outcome of simulations. ML techniques such as support vector machines [17] and artificial neural networks in various forms [18] have been applied to fracture analysis, demonstrating promising performance. The use of high-fidelity fracture network simulations as training data together with novel graph representations was successfully applied to identify areas of high flow and transport [8] in static discrete fracture networks generated by the dfnWorks suite [19]. For the problem of dynamic graph representations of fractures, Miller et al. [20] has employed an image-based approach, extracting video representations of HOSS simulations and using graph convolutional networks to learn features from these. Finally, Moore et al. [21] and Srinivasan et al. [22] have used a variety of ML methods to predict fracture coalescence and time to failure. These studies suggest that important aspects of fracture dynamics can be learned from a modest sample of simulation training data, and then predicted through modern algorithmic techniques.

Fig. 1 summarizes the workflow in our approach: from a continuum model, to HOSS simulations over a statistical ensemble, to the use of ML on dynamic graphs for vastly accelerated predictions of fracture statistics and material failure.

3. Physical system and HOSS simulations

The high-fidelity HOSS simulations, used for training our algorithms, represent a system composed of a rectangular $2\,m \times 3\,m$ 2D concrete sample loaded in uniaxial tension. In order to impose this loading condition, the bottom boundary of the sample is kept fixed while the top boundary is moved with a constant speed of 0.3 m/s, as shown in Fig. 2. This results in a strain rate of $0.1\,s^{-1}$. The material is assumed to be elastically isotropic for all cases. The following elastic material properties were used throughout: density of $2500\,kg/m^3$, Young's modulus of 22.6 GPa, and shear modulus of 9.1 GPa.

Each specimen is seeded with 20 fractures at time $t = 0$, representing preexisting microcracks or defects, each of length 30 cm. With the aim of avoiding overlaps among these initial fractures, the sample was divided into 24 cells arranged in a uniform 4×6 grid, of which 20 cells were selected randomly to contain a fracture. The orientation of each fracture was chosen randomly from one of three angles: 0, 60 and 120 degrees with respect to the horizontal. Finally, each fracture was positioned randomly in its cell, subject to the constraint of being fully contained within the cell.

HOSS simulations use finite elements to model the complex physical process of fracture propagation, with fractures moving along the edges of a triangulated mesh and coalescence points forming at the junctions (Fig. 3 shows an example in the simpler case of only two fractures). The system described above was discretized using approximately 158,000 constant strain triangular elements per specimen. Each vertex of a triangle can move independently in 2D, leading to nearly 10^6 degrees of freedom. Fractures are characterized by a damage value associated with each edge, ranging from 0 (completely intact) to 1 (completely broken). When the damage exceeds a fixed threshold θ, set to $\theta = 0.1$, that edge is considered to be fractured.

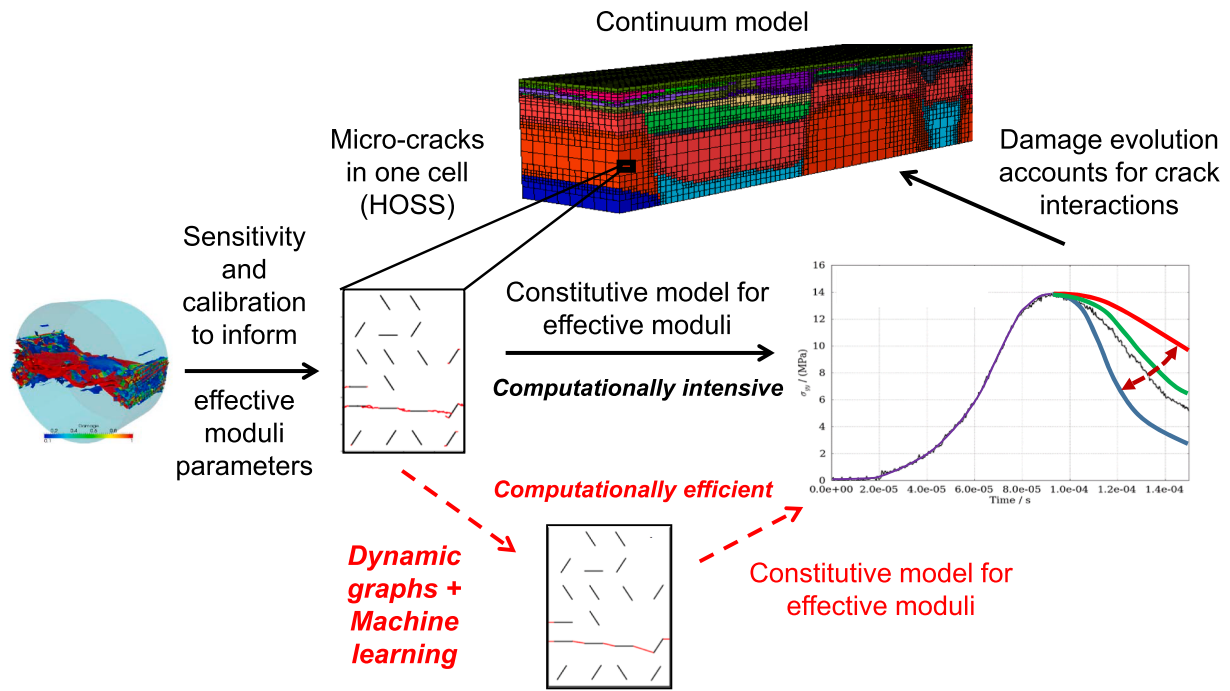Simulations were run for 350 time steps, representing 7 ms of

Fig. 1. Our workflow uses HOSS to provide training data for ML rather than for direct simulation, yielding a computationally more efficient approach to predicting fracture propagation.
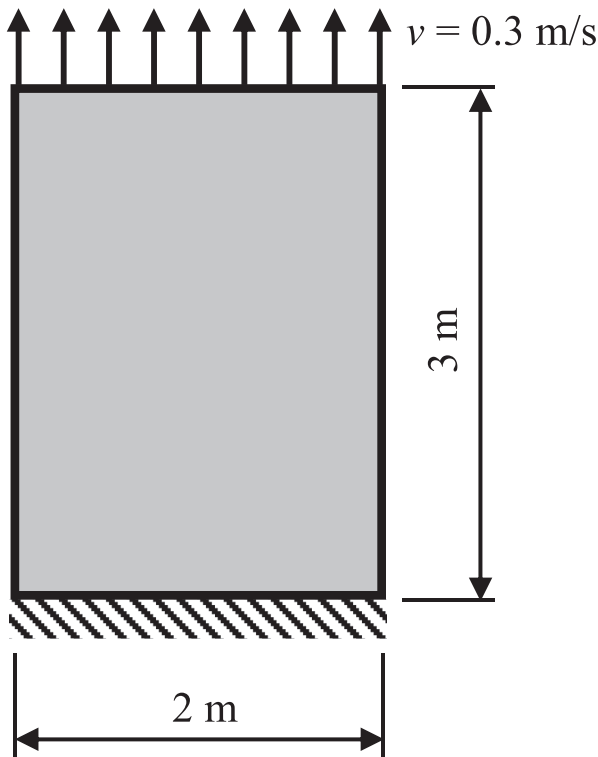

Fig. 2. General setup for the physical model, with boundary and loading conditions as indicated.

fracture evolution. Each simulation took approximately 4 h to run, using 400 processors.

4. Graph formulation

Our approach involves converting the finite element representation to a fracture graph, where vertices represent fractures, labels associated

with each vertex represent fracture attributes (such as length) and edges represent relationships between fractures.

4.1. Fracture graph

We extract the fracture graph by identifying connected components of fractures in the finite element mesh. In order to obtain meaningful predictions of fracture propagation from graph-based neural networks methods, we would like the total number of vertices in the graph to remain fixed as the system evolves and these fractures coalesce. We therefore adopt the following rules:

1. Fracture identity is preserved through coalescence. If two fractures *i* and *j* coalesce at time step *t*, they nevertheless retain their identities at time step *t* + 1. This allows each one to retain its associated features, which will be crucial for our neural network approach. It is important to be able to determine the point of coalescence so that, as the fractures grow, one can decide which finite elements have been newly absorbed into which fractures. This may be ambiguous if fractures grow rapidly between time steps, or if three or more fractures coalesce in one time step. In those cases, we assign the finite element to the fracture where the least additional damage is needed to connect it to its previous elements (see minimal damage distance in Section 4.2.1 below).
2. A finite element associated with a fracture at a given time step remains associated with that fracture for all future time steps. We assume that fractures can only grow over time,

For each simulation, the transformation from finite element data to a fracture graph (see Fig. 4) reduces the data size from about 50 GB to 15 MB, and takes approximately 3 h of processing time. Simulations may be processed in parallel.

Given *N* fractures, we encode the graph structure using a weighted adjacency matrix $A^{N \times N}$, where A_{ij} represents the relationship between vertex *i* and vertex *j*. However, as different types of relationships between fractures can prove important, we use multiple distance metrics (say *d* of them) to describe them, generalizing *A* to a tensor of the form
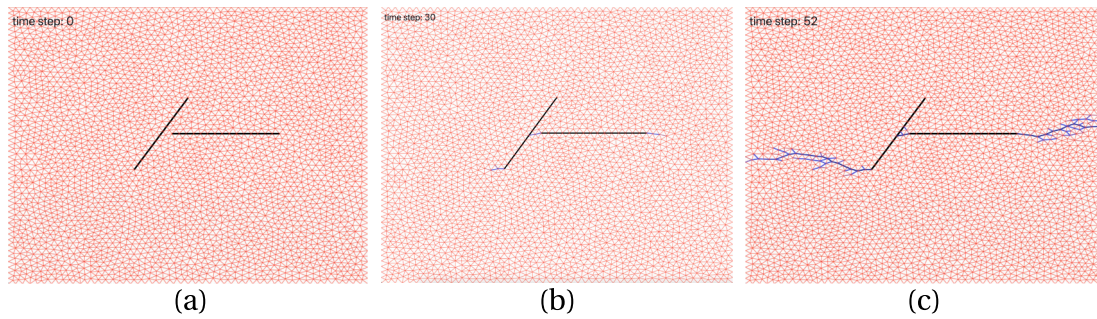
Fig. 3. HOSS triangulation of two-fracture system. (a) Black edges represent initial fracture nuclei. (b) Fractures expand and coalesce. (c) Material failure occurs when fractures connect to boundary.
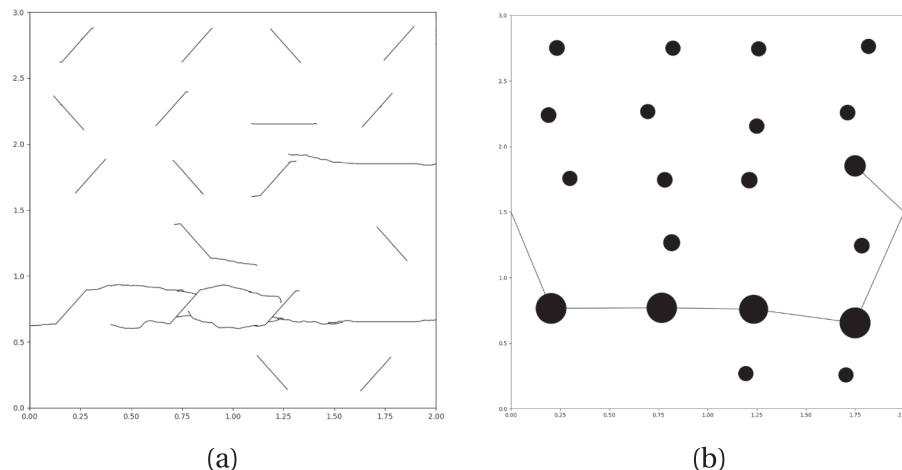
$A^{N \times N \times d}$. Alternatively, we may interpret this as a node-aligned multi-layer network [23] with d separate adjacency matrices $(A_{(1)}^{N \times N}, A_{(2)}^{N \times N}, ..., A_{(d)}^{N \times N})$. Each of these defines a separate graph, but with shared vertex labels, allowing us to process each graph in parallel and then combine the results. We also track a number of different attributes for each fracture, using the matrix $F^{N \times m}$. The d distance metrics and m fracture attributes make up our feature set.

The input to the neural network will be a time series of n graphs $(G_0, ..., G_{n-1})$, where at each time step t, G_t consists of the d adjacency matrices (of size $N \times N$) and the fracture attribute matrix (of size $N \times m$). The output of the neural network will also be in exactly this format, but will consist of predictions; after having seen the first t time steps of the graph, the neural network will predict time step $t + 1$, and so on. We thus train our network by comparing predictions to the true evolution of the fracture graphs. Additionally, this allows a prediction for the evolution of a fracture graph, by iteratively predicting the next state given the starting state and the sequence of previous predictions.

Finally, we define the meaning of material failure in our graph formulation. Failure refers to a system of fractures spanning the system from one boundary to the opposite boundary. Since our fracture graph contains no explicit notion of boundary, we introduce two new vertices representing the boundaries of the material. We then say that the system has failed once there exists a fracture path ("failure path") joining these two vertices.

4.2. Graph features

Next, we describe the features of interest in our graph, associated both with **distances** between fractures and with **attributes** of fractures. We aim to predict the evolution of both categories of features over time, giving information both on individual fracture statistics and on overall damage to a material.

4.2.1. Distance metrics

We use $d = 4$ distance metrics, each of which varies with time:

- **Coalescence.** This is a binary measure of connectivity. A value of 1 represents coalescence of two fractures, while a value of 0 indicates that the fractures remain separate. The quantity is symmetric: $A_{ij} = A_{ji}$ for fractures i and j.
- **Minimal damage distance.** This is the smallest amount of additional damage needed to make two fractures coalesce. Given damage threshold θ, we assign a weight $\max[\theta - \text{damage}, 0]$ to each edge on the finite element mesh, and then find the minimum-weight path needed to link an element from one fracture to an element of the other. This quantity is also symmetric.
- **Oriented tip distance.** We define a fracture tip as a finite element that belongs to the fracture but that is adjacent to only one other element in the fracture. Based on the nearest tip at the previous time step, we approximate the fracture's direction of growth. The oriented tip distance to another fracture is then the projected distance from this orientation to the first point of intersection with that fracture, or a large constant if no intersection occurs. We include this metric on the assumption that fracture tip locations encode important information about the evolution of the fracture system. Unlike the previous two quantities, it is asymmetric.
- **Shortest Euclidean distance.** We define the shortest distance between two fractures as the smallest possible Euclidean distance between an element in one and an element in the other. Although this measure is not necessarily linked to whether two fractures will coalesce, previous studies [24] have noted that simple proximity of two fracture can be an important factor in fracture growth. This quantity is symmetric.

Since the adjacency matrix represent similarities between fractures,

Fig. 4. A fracture system produced by HOSS, and the corresponding fracture graph representation. (a) Lines represent fractures, defined as edges in the finite element mesh with sufficient damage. This fracture network covers a $6 \, \text{m}^2$ area evolving over the course of 7 ms. (b) Nodes represent fractures, with node size signifying total fracture length. Edges represent fracture coalescence.

for the last three metrics we adopt the standard procedure of using a Gaussian kernel to convert distances to similarities:

$$A_{ij} = e^{-\text{dist}_{ij}^2}.$$

4.2.2. Fracture attributes

Our feature set also includes attributes associated with each fracture, and that could be expected to influence fracture propagation. They are represented by the matrix $F^{N \times m}$, where we use $m = 13$ fracture attributes. These attributes describe fracture length, damage, stresses experienced by fracture tips, fracture position (reflecting the observation that fractures often grow faster near the boundaries), orientation with respect to the loading direction, and connectivity.

- **Total damage.** The sum of damage values on all edges associated with the fracture in the finite element mesh.
- **Projected x length.** The projected Euclidean distance of the fracture along the x-axis, which is perpendicular to the loading direction of the material.
- **Projected y length.** The projected Euclidean distance of the fracture along the y-axis, which is parallel to the loading direction of the material.
- **Euclidean length.** The largest Euclidean distance between any two tips of the fracture.
- **Total path length.** The sum of edge lengths associated with the fracture in the finite element mesh.
- **Maximum path length.** The largest sum of lengths of edges between any two tips of the fracture.
- **Max tip stress.** The largest stress at a tip of the fracture. We define the stress at a tip as the average of the Cauchy normal stress values in the y-direction (loading direction), on all finite element edges touching the tip.
- **Mean tip stress.** The stress at a tip of the fracture, averaged over all its tips.
- **Mean fracture x position.** The x-coordinate value of the center of mass of all edges associated with the fracture in the finite element mesh.
- **Mean fracture y position.** The y-coordinate value of the center of mass of all edges associated with the fracture in the finite element mesh.
- **Fracture orientation.** The angle between the loading direction and the line that connects the tip with the smallest x-coordinate value to the tip with the largest x-coordinate value.
- **Fracture degree.** The number of coalescence events experienced by the fracture, equal to the sum of the elements in a row or column of the coalescence adjacency matrix.
- **Global failure.** A binary quantity, replicated as an attribute for every fracture in the system, indicating whether or not material failure has occurred. This quantity reflects the arrest of fracture growth when stress is relieved by failure.

Note that the final attribute is used purely for training purposes, to help the neural network learn connectivity properties, and not for prediction. Material failure is predicted based on the existence of a path of coalesced fractures connecting two boundaries.

5. Machine learning methods

We use deep learning, based on large and diverse artificial neural networks [25], to predict the growth of fractures. Neural networks are a supervised learning method, requiring large amounts of training data. We train on graph representations from multiple simulations of the evolving fracture system. The goal is to predict, given representations of a single simulation at time steps 1, 2, ...,t, the representation at time step $t + 1$. By iteratively applying this method, we can predict all future time steps as well. We generally continue this process until the time step at which material failure occurs.

Deep learning has been applied successfully in the past to graph classification problems [9] and to dynamic graph prediction problems [10]. Many different forms of neural networks exist, and these different forms are frequently used in combination. Convolutional neural network (CNN) models have been used successfully on graphs by using the graph Fourier transform to create a graph convolutional network (gCN) [26]. Kipf and Welling have provided fast and accurate methods for convolutions on large graphs [9], using approximations to the eigen decomposition of the graph Laplacian. Manessi has proposed two promising methods of combining gCNs with recurrent neural networks (RNNs), to exploit graph data with temporal information [10]. The neural network architecture that we use here is made up of several components including feed-forward networks, graph convolutional networks (gCNs), and recurrent neural networks (RNNs).

5.1. Feed-forward neural networks

The basic unit in a neural network is an artificial neuron that receives inputs from a fixed number of sources and decides its output based on a weighted average of its inputs. These neurons are combined as layers, the simplest being a *feed-forward layer* or a *fully-connected layer*, where neurons are independent and process all of the input simultaneously. The resulting output then is primarily a series of matrix multiplications. Given a vector input $X^{m \times 1}$ to the network, the first layer's vector output $Y^{k \times 1}$ may be expressed as

$$Y = \sigma(WX + B).$$

Here, σ is a simple (though generally nonlinear) activation function such as the rectified linear unit, or ReLU, given by $\sigma(x) = \max(0, x)$. W is an $k \times m$ weight matrix, and B is a vector bias allowing the neuron to be activated at an adjustable threshold value. Neural network architectures used in deep learning can consist of many such "hidden" layers, with a feed-forward network representing a composition of activation functions. The key observation is that adding additional layers (depth) to a neural network can allow it to approximate a function far more effectively than adding additional neurons to a given layer [27,28].

The main challenge of a neural network lies in training its adjustable parameters. In the training phase, labeled data are repeatedly passed through the network to determine the error of a given prediction ("loss function"). The weights W and biases B are then adjusted in order to minimize the loss in the training set. Using backpropagation to attribute a specific part of the loss to each neuron, a gradient is calculated, allowing the prediction to be optimized iteratively through a form of stochastic gradient descent.

5.2. Graph convolutional networks (gCN)

A recently developed neural network architecture particularly well suited to operations on graphs [9,29,10] involves using a convolutional layer. The goal of a convolution is to provide spatial invariance: in image processing, for instance, convolving an image with a small kernel can enable recognition of image features regardless of where they are located within the image [30]. Similar methods can be used for graph data. Recall that the input to our neural network is a series of graphs, each defined by an adjacency matrix $A^{N \times N}$ and a node feature (attribute) matrix $F^{N \times m}$. A graph convolutional layer [9] with input F results in the $N \times k$ output

$$Y = \sigma\left(\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\,\widetilde{D}^{-\frac{1}{2}}FW\right),$$

where $\widetilde{A} = A + I$ is the adjacency matrix with added self-connections, $\widetilde{D}$ is a diagonal matrix with $\widetilde{D}_{ii} = \sum_j \widetilde{A}_{ij}$, and W is an $m \times k$ weight matrix. $\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\,\widetilde{D}^{-\frac{1}{2}}$ is the symmetric normalized graph Laplacian, which

takes an average of surrounding vertices and approximates a convolution with the adjacency matrix, so that the graph structure itself forms a filter for the weight matrix. The output Y from the gCN layer consists of k newly computed attributes associated with each of the N nodes in the graph. The use of multiple layers, each with its own weight matrix, allows the use of information from neighborhoods consisting of multiple adjacency steps in the graph.

5.3. Recurrent neural networks (RNN)

RNNs are structurally similar to feed-forward networks, but rather than simply mapping an input X to an output Y, they map a sequence of inputs $X_1, X_2, ..., X_n$ to a sequence of outputs $Y_1, Y_2, ..., Y_n$. Furthermore, they carry forward an internal state, allowing them to learn long-term dependencies between events if the input sequence represents a time series. At time step t, they take the state produced by the previous step, S_{t-1}, together with X_t to produce a new state, S_t, and an output Y_t. The simplest form of RNN uses an activation function

$$S_t = \sigma(W_S S_{t-1} + W_X X_t)$$

and

$$Y_t = \sigma(W_Y S_t),$$

where W_S, W_X, and W_Y are different weight matrices that are adjusted during training. The distinction between a feed-forward network and a RNN may be understood graphically as the network "unrolling" across time (Fig. 5). The internal state S_t represents the RNN's "memory" of the system, constructed from the previous memory together with the new information entering the network.

In practice, the exact operations used to compute S_t and Y_t depend on the RNN implementation. We use the most common method, known as long short-term memory (LSTM) [31]. This includes additional weights allowing states to be "forgotten" while also explicitly using the previous time step's output Y_{t-1} in the calculation at step t. Importantly, RNNs can be incorporated as layers in networks that also includes feed-forward and gCNs, and can be trained using the same methods as these.

5.4. Network architecture

Our architecture combines the components discussed above. The input is a time series of graphs $G_0, ..., G_{n-1}$, each consisting of an adjacency matrix A and a feature matrix F. The output is a series of the same length, but shifted over one time step: $G_1, ..., G_n$, with the network predicting a given G_t based only on the information up to G_{t-1}. This allows us to train the network by comparing the predictions for $G_1, ..., G_n$ with the actual sequence graphs.

Fig. 6 summarizes the procedure for a given time step t. F_t represents the (input) attribute matrix based on fractures, and A_t represents the (input) adjacency matrix based on distance metrics. Following the methods in Manessi et al. [10], the gCN extracts relevant features and the RNN creates a "memory" stored in S_t. These two layers together compute, for each vertex, a set of new features described by the output
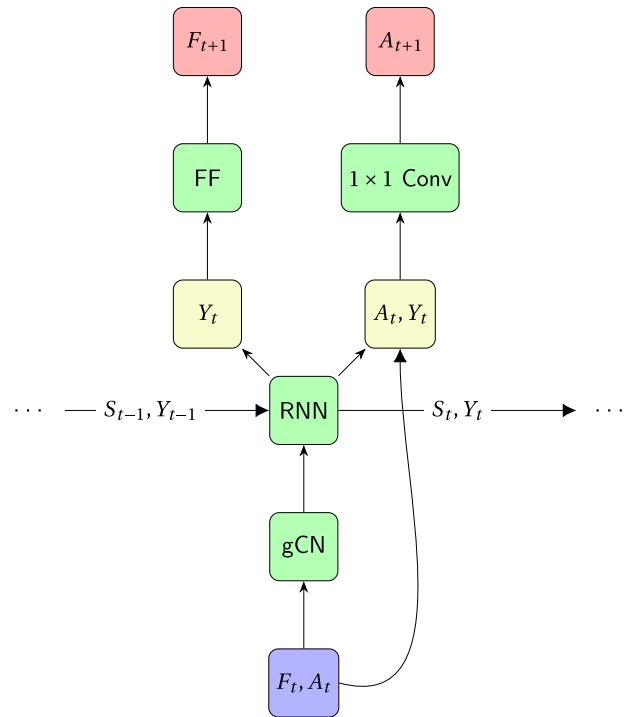

Fig. 6. Outline of one "unrolled" computation section of the algorithm, showing four different neural network processing units: a gCN, an RNN, a feed-forward network, and a 1×1 convolution.

Y_t, taking into account prior developments in the graph. Since we have d different distance metrics, A_t is actually a tensor of dimensions $N \times N \times d$. This is inconsistent with the definition of a graph convolutional layer from [9], which assumes a single $N \times N$ adjacency matrix. We therefore conduct d independent graph convolution operations in parallel and combine the results, ultimately obtaining a concatenated output matrix Y_t of size $N \times kd$, where k is an adjustable parameter corresponding to the number of output features in each graph convolution. We pass Y_t directly into a feed-forward layer to predict the fracture attribute values F_{t+1} at the next time step.

However, predicting the evolution of the adjacency matrix is not straightforward using existing approaches, since gCNs use graph structure to predict vertex features rather than to predict edge features. We therefore augment A_t with the RNN output Y_t. Since the adjacency matrix has dimensions $N \times N \times d$, each element associated with fractures i and j can be considered as a d-dimensional vector. We append to this vector the output features associated with both fractures, creating a new vector of length $d + 2kd$.

We would like to process each of these vectors independently of the others. In order to do so, we apply a 1×1 convolution to the augmented ($N \times N \times d + 2kd$) tensor. This is roughly equivalent to applying a feed-forward layer to each vector, changing its length from $d + 2kd$ to a
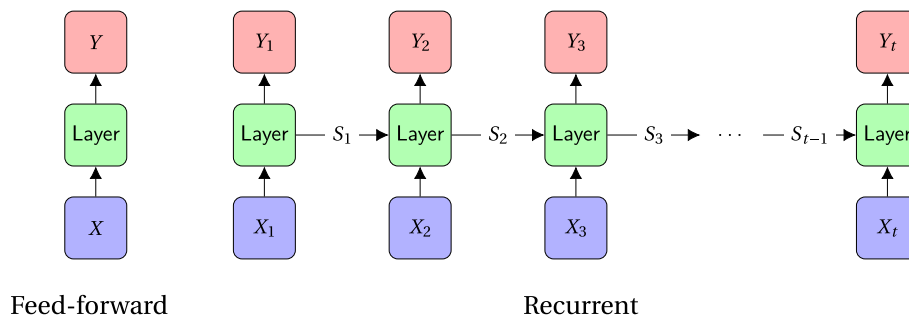

Fig. 5. Feed-forward vs recurrent neural networks.

length of our choosing. By setting the length to d, we recover the correct dimensions for the adjacency matrices. Analogously to the case of fracture attributes, we then use the 1×1 CNN to predict A_{t+1} at the next time step.

Each of the layers in our architecture requires training a set of weights and biases. Consider an adjustable weight w, and a loss function that is differentiable with respect to w. (Note that even if we are trying to predict a discrete quantity such as binary coalescence, the output of the neural network will be a real number that we ultimately threshold, and so an appropriately chosen error will be continuous and differentiable.) There exist both momentum-based algorithms, such as gradient descent, and norm-based algorithms, such as RMSProp, that allow us to update w such that its new value will reduce the loss. We use Nesterov-accelerated Adaptive Moment (Nadam) Estimation [32], which combines standard momentum (using a decaying mean instead of a decaying sum) with RMSProp to improve performance.

We implement the architecture above in Python, using Keras for all components except for the gCN, for which we use the implementation from [9]. In each component, we use two hidden layers, with 64 neurons per layer. Additionally, our use of graphics processing units (GPUs) accelerates training by up to two orders of magnitude as neural network computations, like graphical rendering, consist mostly of large matrix multiplications.

6. Results and discussion

6.1. Training

Our training data consist of 145 HOSS simulations, as described in Section 3. The simulations include time steps 0 through 350 representing the 7 ms of fracture evolution, though some simulations end earlier, right after the material fails. In those cases, we pad the missing time steps by repeating the final state, since fracture systems rarely exhibit any significant change after stress is relieved through material failure.

We train using 5-fold cross-validation. This framework involves partitioning the 145 simulations into 5 subsets, or *folds*, of equal size. We then train 5 separate networks, each one using the data from one of these folds as the test set (29 simulations) and the remaining data as either the training set (101 simulations) or validation set (15 simulations). In this way, collectively over the 5 networks, we ultimately use all of our simulations as test data. For each network, we initialize weights as uniform random variables, and then iteratively update them by training on the 101 simulations, in random order. In each simulation, we sequentially cycle through time steps $t = 0, ..., 349$, inputting graph G_t at each time step. The output is a predicted time series $(G_1^{out}, ..., G_{350}^{out})$ which we compare with the actual graphs. From this comparison, we calculate a loss value as the sum of the mean squared error (MSE) of all features, and use backpropagation to adjust the weights in the network. All real-valued features are rescaled (in preprocessing) to have mean zero and variance one, so they carry equal weight in the loss function. A single training pass on all simulations is called one *epoch*. With every new epoch, a new random permutation of the simulations is generated.

Fracture propagation in HOSS simulations is typically sporadic, with rapid evolution followed by long periods of little or no change. These discontinuities are challenging for a neural network, leading it to overfit by learning only the dormant behavior or learning propagation patterns so abrupt as to be unphysical. To overcome problems such as these, we use three different forms of regularization:

1. **Downsampling.** We use only every 25th time step for both training and prediction, relabeling time steps $0, 25, ..., 350$ as $0, 1, ..., 14$. This has the effect of making fracture evolution more apparent to the network.

2. **Early stopping.** After every 5 epochs, which we call one "iteration,"
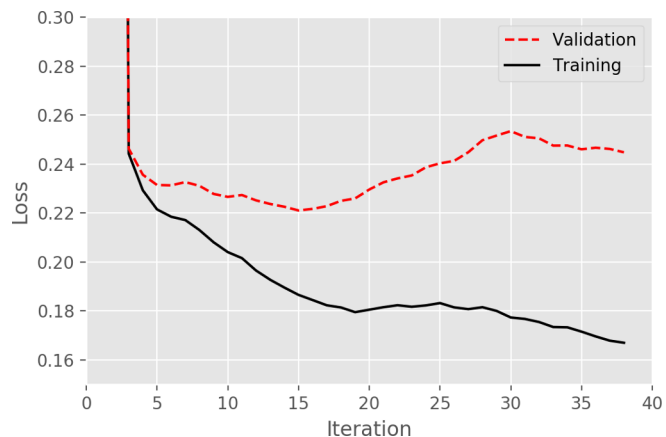
Fig. 7. Learning curve for one network, showing evolution of loss values on training and validation sets (smoothed with moving average using window of width 5 iterations). While training loss is almost always decreasing, validation loss shows marked increase after about 15 iterations, indicating possible overfitting.

we calculate the training error on a separate validation set. If this error starts to increase rather than decrease (see Fig. 7), that signifies possible overfitting to the training set, and so we halt training. Otherwise, we continue training for 40 iterations.

3. **Data augmentation.** The size of our training data is modest by deep learning standards, and further reduced by downsampling. To compensate, we develop an innovative use of synthetic data. After the first iteration, we allow the network to train on its own latest predictions for (downsampled) time steps 1 through 14 of each simulation. We do this by appending those predictions to our real data during the second iteration: we expand the feature set from d distance metrics to $2d$ distance metrics (d of them real, d of them synthetic) and from m attributes to $2m$ attributes (m of them real, m of them synthetic). With each successive iteration, we append an additional set of the most recently generated predictions: after k iterations, the feature set will include $(k + 1)d$ and $(k + 1)m$ features, all of which we use in training.

With the process repeated in this way, the network learns to correct patterns of incorrect prediction, since the loss is always calculated by comparing the output with actual simulation data. In order to decrease the importance of the less accurate earlier predictions, we assign a weight to the loss function contributions that is inversely proportional to the age of the synthetic data: the latest (kth) set has weight 1, the previous set has weight $1/2$, and so on back to the oldest set with weight $1/k$. The improvement in prediction quality due to this method is seen in Fig. 8, showing predicted total fracture damage in one reference simulation (discussed further below). With synthetic data, quantitative predictions track the actual fracture damage far more closely, in terms of both mean and standard deviation.

The entire training procedure, as described above, takes approximately 6 h of processing time on a GPU.

6.2. Prediction

We generate predictions of the entire time series of fracture evolution, based on $t = 0$ conditions alone, for all 145 simulations (each of which is in the test set of one of our 5 trained networks). These predictions are of two kinds: the evolution of fracture sizes, and the global measure of when material failure occurs. On a trained neural network, we typically obtain predictions within seconds.
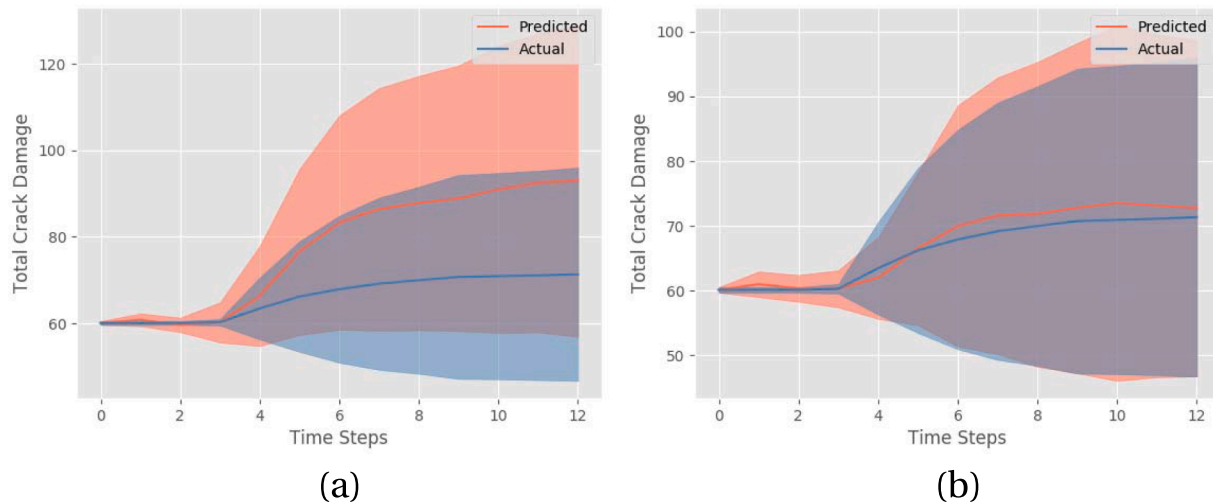
Fig. 8. Evolution of predicted fracture damage vs. actual (i.e., simulated) fracture damage, in one reference simulation, without (a) and with (b) the use of data augmentation. The shaded area indicates one standard deviation from the mean, measured across all fractures in the system.

6.2.1. Fracture growth statistics

One measure of fracture size is the total amount of damage associated with the fracture. Fig. 8 above describes the evolution of predicted and actual (meaning simulated) fracture damage. The mean and standard deviation are taken over all fractures in one reference simulation, with the large standard deviation in part reflecting the concentration of damage among a limited number of fractures. However, recall that there is arbitrariness in the method used to identify individual fractures once they have coalesced (see Section 4.1). Therefore, we also consider damage in connected components of coalesced fractures, focusing specifically on the final time step of a simulation. Fig. 9 shows the distribution of total damage in these coalesced fractures, aggregated over the entire set of 145 simulations. The predicted and actual histograms are both dominated by the peak on the left that represents the overwhelming majority of fractures with only minimal damage. While there is some slight underprediction of coalesced fractures with the least damage, the predicted mean is within 2.45% of its simulated value.

Other measures of fracture size include its length attributes. Fig. 10 shows the predicted evolution of projected x length (perpendicular to the loading direction) of fractures, in the reference simulation used so
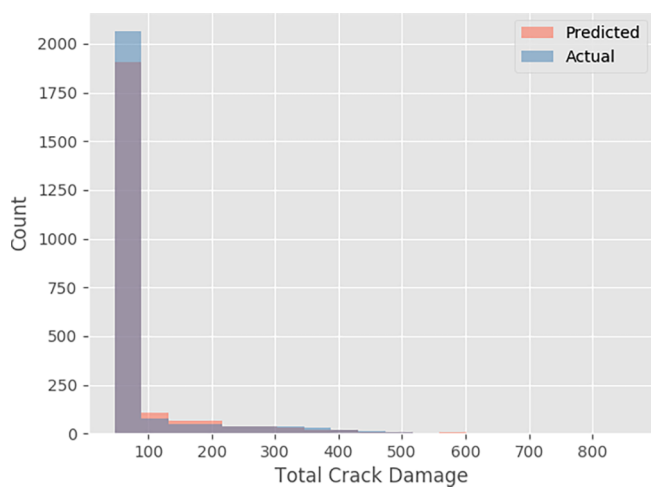
Fig. 9. Distribution of total damage in coalesced fractures (predicted and actual), aggregated over all simulations.

far. Here, the standard deviation is significant even at $t = 0$, since the initial distribution is bimodal, reflecting whether a fracture is oriented horizontally or diagonally (see Fig. 4). Our predictions, while not perfect, agree with many aspects of the actual fracture length evolution, both in terms of overall statistics and in terms of individual fracture behavior.

A length attribute that is directly relevant to material failure but also very closely related to a coalesced fracture's total damage is its total path length. Fig. 11 shows the distribution of this quantity at the final time step, aggregated over the entire set of simulations. Not surprisingly, it strongly resembles Fig. 9, with the predicted and actual histograms again displaying a similar initial peak representing unpropagated fractures. The predicted mean is 1.85% above its simulated value. The difference between the two histograms is described quantitatively by the Kolmogorov-Smirnov (KS) statistic in Fig. 12: KS distances are concentrated at smaller values, with a p-value that lies above 0.05 for 126 out of 145 simulations. Thus, in most cases, the predicted total path length distribution is consistent with the actual one. Surprisingly, this result is robust under ablation, or exclusion of individual features or feature groups during training [33]. We find that the *only* quantity whose removal increases the mean KS distance significantly (from 0.328 to 0.374) is fracture position. This presumably reflects the fact that position can be a determining factor for coalescence, based on whether the fracture is nearer to the boundary or to the center of the material. Conversely, our other features appear to contain enough redundancy that the network can easily make up for the absence of some of them.

6.2.2. Material failure

One of the main strengths of our deep learning approach is that it can provide predictions of global effects such as material failure at the same time as individual fracture statistics. We determine whether a material has failed by whether there exists a connected component of coalesced fractures that includes both boundary fractures. However, a straightforward application of our trained network is not sufficient to obtain accurate results: parameter settings that give accurate predictions of fracture statistics can often predict that the material fails far later than it should.

We therefore modify certain parameters, and also apply a post-processing transformation. The crucial parameters involve a coalescence threshold: since the predicted coalescence adjacency matrix A^{out}
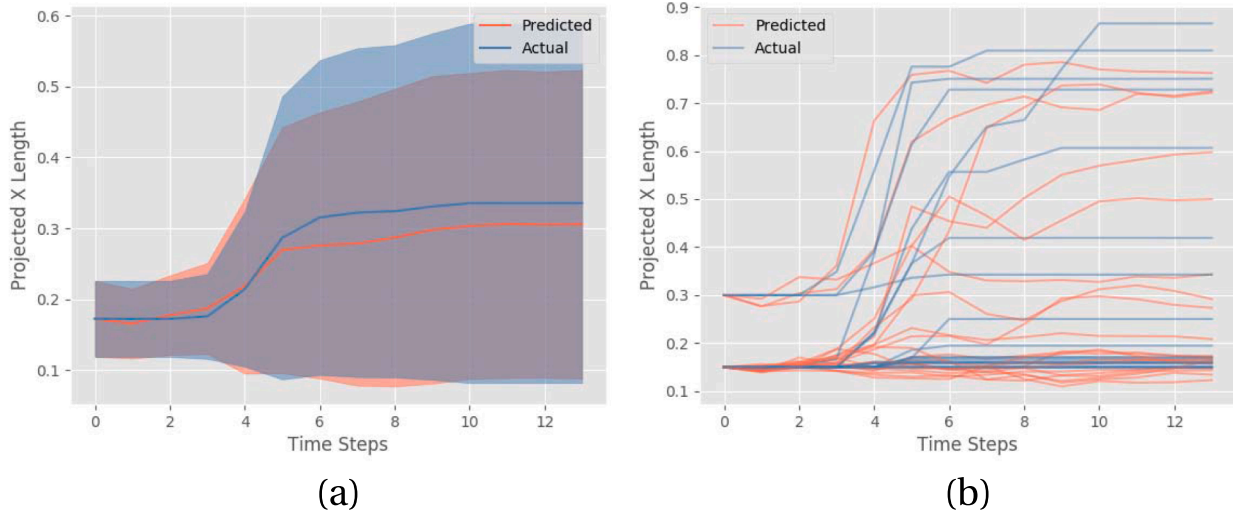
Fig. 10. Evolution of projected *x* length of a fracture (predicted and actual) for the simulation from Fig. 4. In (a), the shaded area indicates one standard deviation from the mean, measured across all fractures in the system. In (b), each individual fracture is plotted.
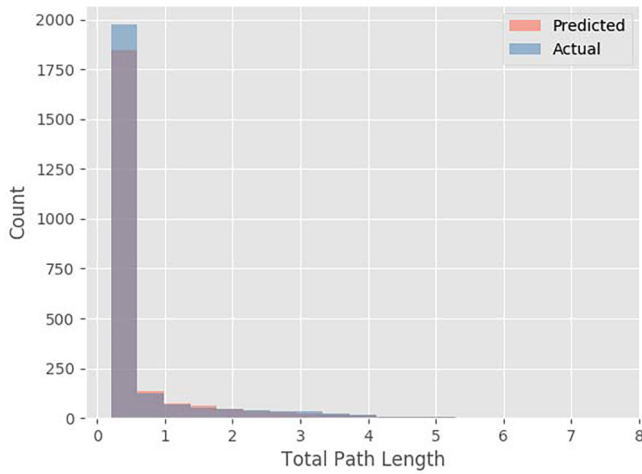

Fig. 11. Distribution of total path length in coalesced fractures (predicted and actual), aggregated over all simulations.

is real-valued, we consider that fractures *i* and *j* have coalesced if either A_{ij}^{out} or A_{ji}^{out} exceed a fixed threshold value. In our previous results, the coalescence threshold for synthetic data used in the data augmentation step of training was set by default to 0.5, and the threshold used in prediction was set to 0.1. Raising the training threshold makes coalescence prediction more aggressive, as it pushes the network to correct its perceived underprediction in the data augmentation process. By contrast, raising the prediction threshold makes coalescence prediction more conservative. We find that raising both thresholds simultaneously provides the best results: we set the training threshold to 0.9 and the prediction threshold to 0.5.

Of the original 145 simulations on which we make predictions in our 5 trained networks, 84 of them do not exhibit failure by the final time step. In those cases, actual (simulated) time to failure is undefined, so we omit them from our analysis and only obtain predictions for the remaining 61 simulations. When predicting time to failure, we allow the network to run beyond time step 14 if necessary. We then perform an additional postprocessing step using ridge regression (Tikhonov regularization) and leave-one-out cross-validation, where for each one of the 61 simulations, we learn a separate transformation, based on the
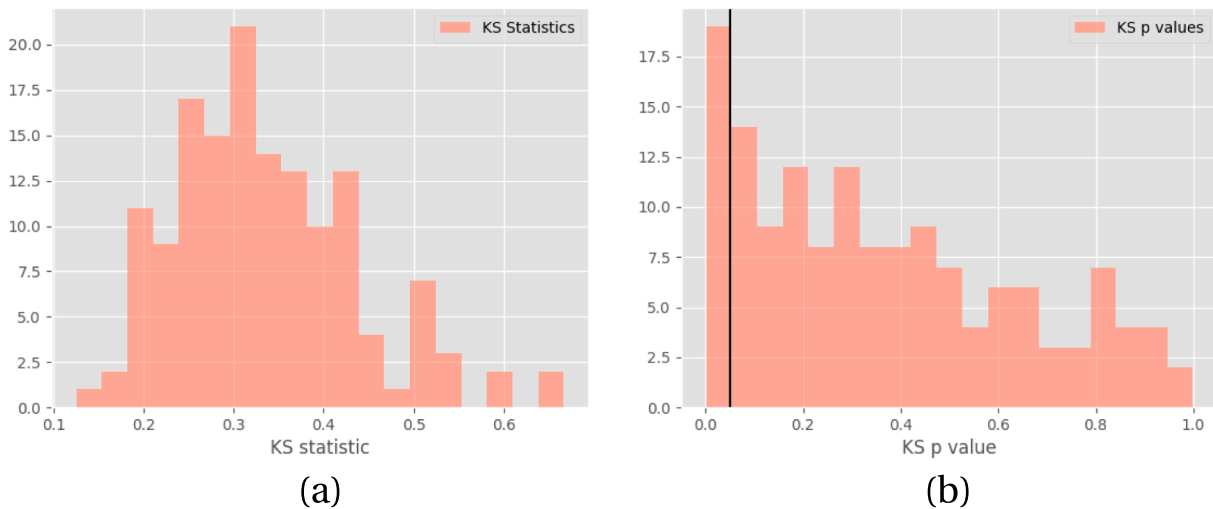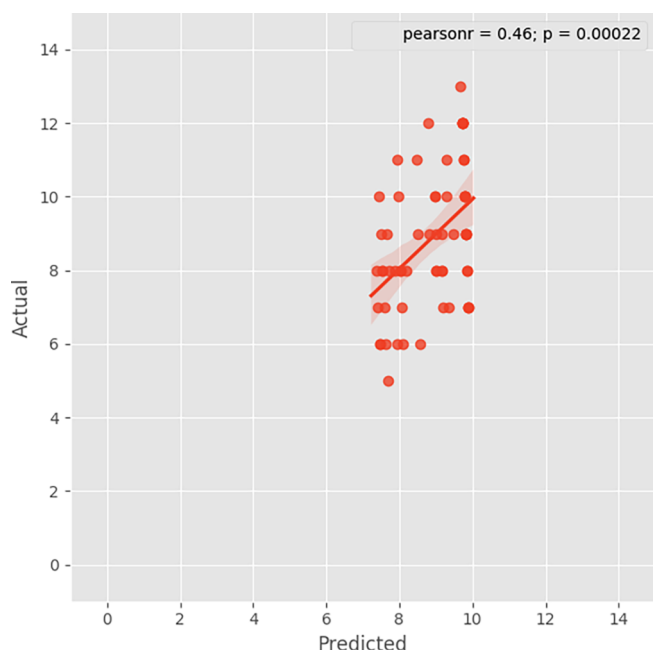

Fig. 12. Histogram, over all simulations, of (a) the KS statistic measuring the difference between predicted and actual distributions of total path lengths in coalesced fractures, and (b) the *p*-value corresponding to the KS statistic. Vertical line shows value above which distribution difference is not significant at a level of $\alpha = 0.05$.

Fig. 13. Scatter plot of predicted versus actual time to failure, for the 61 fracture systems that exhibit material failure during the simulation. MAE is 15.4%.

actual times to failure and the network predictions for the 60 other simulations. In principle, this method could entail a small amount of data leakage: the transformation is learned from results from all 5 trained networks, 4 of which were trained on data that included the test simulation itself. However, given that a network is trained on a set of 101 simulations, it appears unlikely that one single simulation could influence the results enough to bias the transformation measurably.

Fig. 13 shows a scatter plot with predicted vs. actual time to failure for the 61 fracture systems, after applying the postprocessing transformation. The mean absolute error (MAE) in the prediction is 15.4% of the average time to failure, and the correlation coefficient is $r = 0.46$. This result is competitive with the most recent ML approaches [21] based on similar HOSS data, which have an error of approximately 16% and a correlation coefficient varying from $r = 0.42$ to $r = 0.68$, even though those methods (unlike ours) are specifically oriented towards predicting failure paths. The MAE in our results reflects the non-negligible scatter in predicted values for a given actual time to failure, which may arise in part from our features imperfectly characterizing the geometry of fractures. While this issue would not impact predictions of fracture growth statistics, which directly measure the feature values themselves, it could easily cause imprecision in predicting higher-order effects such as material failure. Our postprocessing will correct any systematic overestimate or underestimate in predicted time to failure (see regression line in Fig. 13), but it will not reduce the variability in prediction. It is worth noting, however, that this phenomenon is hardly unique to our approach. Time to failure is notoriously difficult to predict even with high-fidelity models: the quantity exhibits large sample-to-sample variations in the original HOSS simulation data [6], even when other fracture growth statistics are relatively stable.

7. Conclusions

We have presented a novel ML approach that provides rapid and accurate predictions of fracture propagation in brittle materials. Using deep learning, we construct a neural network architecture that combines a gCN with an RNN, and train the network on high-fidelity simulations from the HOSS model. We compensate for the relatively modest size of our training data set with a new form of data augmentation, training the network not only on simulation results but also on incorrect predictions that it has itself generated in earlier training passes. In this way, the network learns to correct its own mistakes, and successfully predicts how properties of the fracture system evolve from their $t = 0$ state alone. A significant benefit of an ML approach is speed: whereas a single HOSS simulation takes hours to run, our neural network generates results in seconds once it has been trained. This speed consideration is particularly crucial in the context of uncertainty quantification. HOSS simulations themselves represent random realizations of a statistical ensemble, and thousands of such simulations could be required to produce accurate estimates of material behavior.

Our fracture size predictions, which we test using 5-fold cross-validation, are in good agreement with the results of simulations. The predicted total damage in a coalesced fracture at the final time step, averaged over all 145 simulations, is within 2.95% of its simulated value. The predicted total length of a coalesced fracture is within 1.85% of its simulated value. In 87% of the simulations, the difference between the predicted and simulated length distribution is not statistically significant, at the significance level of a Kolmogorov-Smirnov test. While other ML approaches have also proven successful for modeling fracture dynamics, deep learning has the advantage that it can generate predictions simultaneously for qualitatively distinct material properties. In particular, we predict time to material failure with an accuracy that is competitive with the state-of-the-art. The MAE in our predictions is approximately 15% of the average time to failure, with a correlation coefficient of 0.46 between simulated times and predicted times.

It is important to note that the results presented in this work are based on high-fidelity models that have been run under a given type of boundary and initial conditions. If different conditions were to be addressed, then the process would need to be repeated. Ideally, one would expect to devise a matrix or a space of boundary and initial conditions that would cover the most relevant situations of interest for a particular problem. This would then create a library of high-fidelity simulation results used to inform machine learning algorithms, for predicting more general loading scenarios.

Finally, we mention possible directions for improving the quality of results. Predicted fracture sizes are stable under exclusion of different groups of fracture attributes, suggesting redundancy in the current choice of features. The use of new features, representing different physical properties, may improve predictions. Additionally, time-to-failure predictions could possibly be improved by the use of ensemble learning, combining the results of multiple trained networks with differently adjusted parameters. This is due to a significant class imbalance when predicting the coalescence of individual fractures: only a small minority of fracture pairs ever coalesce. Preliminary studies suggest that these pairs may be more accurately identified by using an ensemble of networks trained with different coalescence threshold settings, with an additional supervised learning step to map their multiple predictions onto a single one. Alternatively, a reweighting scheme could be used in such cases [21], so that a classifier has adequate opportunities to learn cases of coalescing fractures. Implementing class reweighting could potentially improve coalescence predictions to the point where a single model would provide highly accurate results for time to failure.

Data availability

The raw/processed data required to reproduce these findings cannot be shared at this time due to technical or time limitations.

CRediT authorship contribution statement

Max Schwarzer: Conceptualization, Methodology, Software, Validation, Investigation, Writing - original draft, Writing - review & editing, Visualization, Supervision. **Bryce Rogan:** Methodology, Software, Validation, Investigation, Writing - original draft, Visualization. **Yadong Ruan:** Methodology, Software, Validation, Investigation, Visualization. **Zhengming Song:** Methodology, Software, Validation, Investigation, Writing - original draft, Visualization. **Diana Y. Lee:** Methodology, Software, Investigation, Writing - original draft, Supervision. **Allon G. Percus:** Conceptualization, Methodology, Validation, Investigation, Writing - original draft, Writing - review & editing, Visualization, Supervision, Project administration. **Viet T. Chau:** Software, Data curation. **Bryan A. Moore:** Software, Data curation. **Esteban Rougier:** Conceptualization, Software, Writing - review & editing, Visualization. **Hari S. Viswanathan:** Conceptualization, Validation, Visualization, Supervision, Project administration. **Gowri Srinivasan:** Conceptualization, Validation, Visualization, Supervision, Project administration.

Acknowledgments

References

[1] W. Fleming Petri, D. Kuhl, Static and Dynamic Crack Propagation in Brittle Materials with XFEM, Schriftenreihe Fachgebiet Baumechanik/Baudynamik, Kassel University Press, 2013.

[2] J.D. Hyman, J. Jiménez-Martínez, H.S. Viswanathan, J.W. Carey, M.L. Porter, E. Rougier, S. Karra, Q. Kang, L. Frash, L. Chen, Z. Lei, D. O'Malley, N. Makedonska, Understanding hydraulic fracturing: a multi-scale problem, Philos. Trans. Series A Math. Phys. Eng. Sci. 374 (2078) (2016) 20150426.

[3] J. Kim, E.S. Um, G.J. Moridis, Fracture propagation, fluid flow, and geomechanics of water-based hydraulic fracturing in shale gas systems and electromagnetic geophysical monitoring of fluid migration, SPE Hydraulic Fracturing Technology Conference, Society of Petroleum Engineers (2014).

[4] P. White, Review of Methods and Approaches for the Structural Risk Assessment of Aircraft, Tech. Rep. DSTO-TR-1916, Australian Government Department of Defence, Defence Science and Technology Organisation, 2006.

[5] A. Hillerborg, M. Modéer, P.-E. Petersson, Analysis of crack formation and crack growth in concrete by means of fracture mechanics and finite elements, Cem. Concr. Res. 6 (6) (1976) 773–781.

[6] E. Rougier, E.E. Knight, S.T. Broome, A.J. Sussman, A. Munjiza, Validation of a three-dimensional finite-discrete element method using experimental results of the split Hopkinson pressure bar test, Int. J. Rock Mech. Min. Sci. 70 (2014) 101–108.

[7] J.N. Vevatne, E. Rimstad, S.M. Hope, R. Korsnes, A. Hansen, Fracture networks in sea ice, Front. Phys. 2 (2014) 1–8.

[8] M. Valera, Z. Guo, P. Kelly, S. Matz, V.A. Cantu, A.G. Percus, J.D. Hyman, G. Srinivasan, H.S. Viswanathan, Machine learning for graph-based representations of three-dimensional discrete fracture networks, Comput. Geosci. 22 (3) (2018) 695–710.

[9] T.N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, 2016. arXiv:1609.02907.

[10] F. Manessi, A. Rozza, M. Manzo, Dynamic graph convolutional networks, 2017. arXiv:1704.06199.

[11] A. Munjiza, D. Owen, N. Bicanic, A combined finite-discrete element method in transient dynamics of fracturing solids, Eng. Comput. 12 (2) (1995) 145–174.

[12] A. Munjiza, The Combined Finite-Discrete Element Method, Wiley, 2004.

[13] Z. Lei, E. Rougier, E.E. Knight, A. Munjiza, A framework for grand scale parallelization of the combined finite discrete element method in 2d, Comput. Particle Mech. 1 (3) (2014) 307–319.

[14] J.W. Carey, Z. Lei, E. Rougier, E.E. Knight, H. Viswanathan, Shale fracture analysis using the combined finite-discrete element method, AGU Fall Meeting Abstracts, American Geophysical Union, 2014.

[15] K. Okubo, H.S. Bhat, Y. Klinger, E. Rougier, Modelling earthquake ruptures with dynamic off-fault damage, in: EGU General Assembly Conference Abstracts, vol. 19 of EGU General Assembly Conference Abstracts, 2017, p. 4992.

[16] H. Djidjev, D. O'Malley, H. Viswanathan, J. Hyman, S. Karra, G. Srinivasan, Learning on graphs for predictions of fracture propagation, flow and transport, 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).

[17] G.-R. Shi, Superiorities of support vector machine in fracture prediction and gassiness evaluation, Pet. Explor. Dev. 35 (5) (2008) 588–594.

[18] H. Wang, W. Zhang, F. Sun, W. Zhang, A comparison study of machine learning based algorithms for fatigue crack growth calculation, Materials 10 (5) (2017) 543.

[19] J.D. Hyman, S. Karra, N. Makedonska, C.W. Gable, S.L. Painter, H.S. Viswanathan, dfnWorks: a discrete fracture network framework for modeling subsurface flow and transport, Comput. Geosci. 84 (2015) 10–19.

[20] R.L. Miller, B. Moore, H. Viswanathan, G. Srinivasan, Image analysis using convolutional neural networks for modeling 2d fracture propagation, in: 2017 IEEE International Conference on Data Mining Workshops (ICDMW), vol. 1, 2017, pp. 979–982.

[21] B.A. Moore, E. Rougier, D. O'Malley, G. Srinivasan, A. Hunter, H. Viswanathan, Predictive modeling of dynamic fracture growth in brittle materials with machine learning, Comput. Mater. Sci. 148 (2018) 46–53.

[22] G. Srinivasan, J.D. Hyman, D.A. Osthus, B.A. Moore, D. O'Malley, S. Karra, E. Rougier, A.A. Hagberg, A. Hunter, H.S. Viswanathan, Quantifying topological uncertainty in fractured systems using graph theory and machine learning, Sci. Rep. 8 (2018) 11665.

[23] M. Kivelä, A. Arenas, M. Barthelemy, J.P. Gleeson, Y. Moreno, M.A. Porter, Multilayer networks, J. Complex Networks 2 (3) (2014) 203–271.

[24] R. Tchoua, Data-driven Approach to Predicting Crack Propagation in Brittle Materials, Los Alamos National Laboratory, 2017 Tech. rep..

[25] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (7553) (2015) 436–444.

[26] M. Henaff, J. Bruna, Y. LeCun, Deep convolutional networks on graph-structured data, 2015. arXiv:1506.05163.

[27] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, Neural Networks 2 (5) (1989) 359–366.

[28] H.W. Lin, M. Tegmark, D. Rolnick, Why does deep and cheap learning work so well? J. Stat. Phys. 168 (6) (2017) 1223–1247.

[29] M. Defferrard, X. Bresson, P. Vandergheynst, Convolutional neural networks on graphs with fast localized spectral filtering, 2016. arXiv:1606.09375.

[30] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, Adv. Neural Inf. Processing Syst. (2012) 1097–1105.

[31] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780.

[32] S. Ruder, An overview of gradient descent optimization algorithms, 2017. arXiv:1609.04747.

[33] A. Nguyen, J. Yosinski, J. Clune, Deep neural networks are easily fooled: High confidence predictions for unrecognizable images, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 427–436.