Imperial College London

Applied Computational Science and Engineering

Independent Research Project

Implementing integrated machine learning strategies to accelerate high accuracy fracture growth simulators

John C N W Walding

29/08/2019

Email: waldingjc@gmail.com

Github: waldingjc

Supervisor: Dr. Adriana Paluszny Rodriguez

Contents

Fractures | Simulation | Machine Learning | Neural Network | Python

Abstract

Current fracture simulation relies upon Finite-Discrete Element Models, which calculate fracture evolution by implementing the known mechanics of fracture dynamics. Whilst accurate, this method is often slow due to the necessity of the solution of several partial differential equations at each time step. This is especially apparent for large systems.

Therefore, an accelerator is proposed that leverages Machine Learning strategies in order to avoid the computation of the partial differential equations, by instead training a neural network on previously run simulations. The specific approach is to use positional, geometric and historical data as features for a feedforward neural network in order to predict the stress intensity factor at a given fracture tip.

The results from the final runs of the solution returned average errors of 5.11%, 21.6% and 18.0% for each component of the stress intensity factor (K_I, K_{II} and K_{III}) respectively, after a run time for 1000 epochs.

These results show the viability of such an approach, though further iteration is required until this strategy can be successfully incorporated into mature simulators. A particular avenue for further development is better integration of fracture interaction data.

1 Motivation

The study of the evolution of fractures in brittle solids is critical to many current fields of research, from hydraulic fracturing and mineral extraction, where a knowledge of how fractures grow is essential for system efficiency, to the study of fractured and broken bones for medicinal purposes, where understanding how a fracture may evolve can help inform an optimal treatment plan.

Furthermore, due to the omnipresence of brittle solids in all aspects of life, an increased capacity to predict fracture mechanics is of value to countless is invaluable to generic engineering, construction and product design applications.

Current fracture growth and propagation analysis is performed using Finite-Discrete Element Models (FDEMs) such as the Hybrid Optimization Software Suite (HOSS) (Knight, et al., 2014) or the Imperial College Geomechanics Toolkit (ICGT) (Paluszny, et al., 2007) (Paluszny & Zimmerman, 2011) (Paluszny, et al., 2013). These require fine spatial resolution on the cracks, and then perform forward time evolution in accordance to Newtonian mechanics, fluid flow and thermal effects, among other factors.

Whilst accurate, these models often take prohibitive amounts of time to complete due to the requirement for the solution of several Partial Differential Equations (PDEs) (Paluszny, et al.,

2018). Furthermore, due to the infeasibility of knowing the exact distribution of cracks and microcracks in a given sample, for statistical rigor several models have to be run with perturbations on the initial state, further compounding the time inefficiency of these models (Moore, et al., 2018).

As a result of these factors, there is room for improvement with regard to the computational efficiency of these simulators. With this increase in computational efficiency, the amount of system perturbations modelled can be increased for the same quantity of time, increasing the statistical validity of any findings.

2 Introduction

The application of Machine Learning (ML) strategies to formerly mechanically based fields is becoming more common, with the goal often being an increase in computational efficiency. There have been studies into the application of ML to fracture analysis already, with measured success. For example, (Hunter, et al., 2019) has the most in depth discussion of alternate approaches to the simulation of fracture propagation. The paper designs an abstraction of the system in order to apply these approaches. The abstraction restricts the domain to two dimensions, populates the domain with a number of one dimensional cracks at one of three orientations, and only considers Mode I (opening) damage. The system has a constant tension applied vertically, and the time to failure is predicted, where failure is defined when a contiguous crack spans the horizontal domain. This system is shown in schematic form in Figure 1.

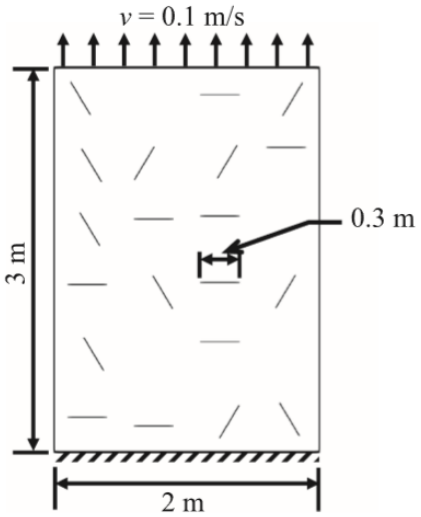


Figure 1: A schematic representation of the abstraction designed in (Hunter, et al., 2019) Source: (Hunter, et al., 2019)

After constructing this system, the authors then proceeded to test a number of graph theoretic models for fracture evolution and evaluated their success by comparison of the predicted time to failure from the model to that which was predicted by HOSS, a fully formed simulator. Certain approaches, especially the Micro-crack Pair Informed Coalescence (McPIC) strategy, achieved some level of correspondence with the HOSS predictions, despite the massively abstracted view of the system. Some related and follow-up papers have a similar approach, and again achieve some success with respect to the authors' assigned metrics.

However, these studies are limited to graph theoretic architectures and often a two dimensional surface with one dimensional fractures. Whilst effective at demonstrating the viability of an ML approach and possibly serving as a foundation for further development, the methodology proposed therein is, at time of writing, not directly applicable to real world problems.

The objective pursued in this paper is to develop a method that leverages ML strategies in such a fashion that it improves or assists current state-of-the-art simulators for immediate utility without requiring the construction of an entirely novel framework. In order to do this, one must have some understanding as to how these current simulators operate – discussed in Section 3.1. This investigation illuminates two notable avenues for ML acceleration, the first of which being the calculation of the Stress Intensity Factor (SIF).

The SIF is made up of three elements, K_I, K_{II} and K_{III}, representing opening damage, in plane shear damage and out of plane shear damage respectively. The calculation of these quantities is a pre-requisite to the calculation of fracture tip evolution. Unfortunately, this calculation is computationally demanding, requiring the solution of several PDEs. The first proposal is therefore to circumvent this calculation by training a neural network to predict these quantities, and then use said trained neural network instead of the calculations currently required, the intention of which is to dramatically reduce the computational time required whilst only resulting in minimal loss of accuracy.

The second possible avenue is predicated on the fact that computational power is finite, and must therefore be allocated intelligently where possible. Currently, the geometric density of the ICGT is constant throughout the domain, which inexorably leads to one of two negative outcomes. Either the density is uniformly high, such that accuracy is maintained even when fracture boundaries approach one another, at the cost of often unnecessary resolution slowing execution, or the density is uniformly low, leading to often adequate resolution in the generic cases, but with an inability to accurately compute situations where two fractures approach to within this resolution. If one was able to intelligently increase and decrease geometric density, one would be able to avoid both of the aforementioned pitfalls. In order to achieve this, a neural network could be trained to predict when two fractures are within substantial interaction range by observing the affect two fractures have upon one another's SIFs. This is the second proposal.

After the implementation of one or both of these proposals, they will be incorporated into the ICGT in order to determine whether they are successful at retaining accuracy whilst increasing performance.

3 Background

3.1 Fractures

While a deep understanding of fracture dynamics is not necessary to follow this project, a very brief overview is included to provide context. The problem being considered is how to predict the growth of fractures in brittle media. This is an inherently complex problem, due to the quantity of overlapping factors that influence it, including the elasticity and plasticity of the medium, the thermal properties of the medium, the micro- and macro-scale structure of the medium and the nature of the applied tension to the body, not to mention the additional complications arising from the possible presence of a fluid. Despite the clear

difficulty in doing so, mathematical frameworks have been developed in order to quantify this behaviour, though the intricacies of said frameworks are beyond the scope of this review.

Broadly however, a simulator implementing one of said frameworks will often take the form of a Finite Discrete Element Model (FDEM). This involves the discretisation of the medium into a conglomeration of geometric objects and the discretisation of time into steps. After which one can perform calculations on the current state of the medium using the framework, and then iterate one step into the future, and repeat for as long as one requires. The fractures in such a simulator are often represented as collections of fracture tips, the evolution of which are considered individually, and together show the evolution of the fracture as a whole. As a gross simplification of how the evolution of a tip is calculated; first, the displacement vector at the tip is calculated using the information held within the system, this is then used to calculate the Stress Intensity Factor (SIF) at the tip, which is then finally used to calculate the spatial evolution of the tip. The final step in this chain is well understood and not computationally expensive, the second step is similar but to a lesser degree, whilst the first step is very complex and considers all the factors mentioned above and more. The objective of this project is to be able to predict the SIF without having to perform the preceding computations, and therefore avoid the most computationally intensive aspects of the simulation.

As it is the prediction target, the SIF bears some further consideration. As mentioned prior, the SIF consists of three elements, K_I, K_{II} and K_{III}, representing opening, in plane shear and out of plane shear respectively. The nature of common loading configurations leads to the primacy of K_I over the other two, as fractures are often aligned close to perpendicularly with the load. Despite their possible secondary nature, K_{II} and K_{III} are still highly important, especially in complex multi-fracture systems where the local tension can differ significantly with the global tension. This is problematic as their more complex mechanics result in heightened difficulty in their prediction.

3.2 Machine Learning

Before discussion of the specifics of the solution, a brief aside to give background on Machine Learning, the core of the project. Starting in the late 50s to early 60s, ML began to develop as a field of research, primarily consisting of *perceptrons*, which are early neural networks, and are similar to that which is implemented in this project. The initial concept is a development of simple linear regression, where one attempts to predict future values when given a dataset, assuming that said future values and the dataset are drawn from the same distribution. Whilst a powerful tool, linear regression is limited to linear systems, as the name implies. However, the same principle can be extended by introducing further degrees of freedom. In practice this is implemented by introducing “hidden layers” between the input and output, where these hidden layers consist of neurons. These neurons are then “activated” in proportion to the sum of all the neurons in the preceding layer, where each connection between a neuron in the previous layer and the current neuron is weighted. These weights are the quantities

that can be changed in order to alter the network's output. A perceptron can be reduced to linear regression by using only one hidden layer with only one neuron.

Whilst this increases the predictive power of a perceptron beyond that possible with linear regression, this network approach is not useful in isolation, as the weights would have to be manually tuned to give the correct outputs. This operation quickly becomes unfeasible with increasing neuron density. The breakthrough required to make full use of a neural network is *backpropagation*. This is an iterative, recursive algorithm that can calculate the optimal changes to the weights by analysing the end result of the network with respect to some predefined loss function (or criterion function). The loss function returns a real number that represents the "loss" of the network, or the degree to which the output of the network diverges from the expected result. With the incorporation of backpropagation, a perceptron can "teach" itself the optimum weights for the given problem, without the guidance of a human operator. The power of this approach is codified in the Universal Approximator Theorem (Cybenko, 1989), which states that any function can be arbitrarily well approximated by a sufficiently complex network with a sufficiently large training dataset.

While the field of Machine Learning has progressed beyond this (e.g. convolutional neural networks and generative adversarial networks), the strategies employed herein are based off the theory described above.

4 Software Development Life Cycle

4.1 Development Overview

The broad development strategy employed for this solution was an iterative paradigm, where after the problem has been analysed, a relatively prescriptive design will be drafted. Once completed, the design will be implemented as a prototype, which is then tested and evaluated against the points raised during analysis. The design will then be amended with the knowledge gained from the prototype, and the cycle will begin anew. This paradigm was used as it allows the incorporation of knowledge gained during the development process, similar to agile development, whilst also providing more structure to the process, as in the waterfall approach. It also avoids the pitfalls to both of these paradigms, as the iterative structure avoids the rigid determinism of waterfall whilst the ever changing but present design restricts the tendency for code developed under an agile framework to be unsustainable. Each of the stages in this iterative paradigm are discussed below.

4.2 Problem Analysis

As described in the introduction, two proposals have been raised. It was determined that the prediction of the SIF would be more useful as a standalone solution and was therefore developed first. Given this objective, there are two evaluation factors above all others; accuracy and efficiency. Without either of these the solution would be unsuccessful. However, given that the nature of the current calculation of the SIF is inherently time-

consuming due to the requirement of the solution of PDEs, it is expected that any ML based solution will be more time efficient. As a result, the primary goal of the solution is prediction accuracy, although as this is a first implementation, demonstration that such a solution is possible as a proof of concept is also an objective.

Furthermore, as the intention for the solution is to be integrated into the ICGT as an accelerator, it should be designed to be as modular as possible, such that it can be inserted with little difficulty.

4.3 Solution Design

When one is designing an ML system, the first decision to be made is which strategy to employ, as it affects every further decision to be made. The canonical solution would be to have a list of the prior SIFs for the fracture tip and use linear regression in order to predict the next SIF (which in reality is barely an ML strategy at all). However, this is a very simplistic strategy and ignores the physical system altogether. Primarily, there can be no expectation that a timewise examination of a given fracture tip's SIF even has a relationship to discern, as the behaviour of the SIF is entirely dependent on the current system configuration and has very little memory for prior configurations beyond the fact that a new fracture is highly unlikely to be where the tip in question once was. Therefore, a more complex strategy must be considered, one that is more dependent on the current state.

In the design phase, the implementation of a Convolutional Neural Network (CNN) was considered, as a CNN would inherently consider the geometric structure of the system, which is a large influence on the SIF. The concept was to create a matrix centred on the tip in consideration, of size equal to the radius of influence of the tip (this radius of influence is a known quantity in fracture mechanics already). The values therein would then be the current SIFs of any tips inside that radius, geometrically located within the matrix where their respective tips are located with respect to the tip under consideration in the domain. However, the architecture of the ICGT does not lend itself to analysis by a CNN, as the data such a CNN would require as a dataset is not readily available (primarily the lack of an attached history for each tip). Furthermore, as the SIF consists of three elements and the system is in three dimensions already, the input to the CNN would have to be in 27 dimensions, which was determined to be unfeasible for a first attempt at the problem.

The other possibility considered, and the one that was developed, is a simple linear neural network. As it takes its input as a set of floats, any number of features can be used, and there is no limit on what those features can be so long as they can be represented as one or more floats. This also detaches the model from any direct interaction with the simulator at the training phase, as the data used for training can be saved and accessed simply. This is in contrast to the CNN concept where the specific geometry of the simulator would have to be considered. It is for these reasons the neural network was chosen as the approach for this project.

For a neural network, the most important factor is often the dataset itself, and therefore one that warrants significant consideration. As the training data for this solution is drawn from simulations run by the ICGT, there is the liberty of designing one's own dataset. In order to construct the most efficient feature set for this problem, one must consider the mechanics of fracture simulation as they are currently understood.

As discussed in relation to a simple regression approach, fracture evolution is determined nigh exclusively by the current configuration of the system, and therefore the feature set should reflect this. This manifests in the exclusion of historical data from the feature set, leaving only the parameters of the current system. Of these, an obvious selection is the current SIF at the tip. Unlike historical SIF data, the current SIF may well be an indicator for the SIF one iteration into the future, as with fine enough temporal resolution the situation surrounding the tip is unlikely to change significantly between iterations. Another obvious selection is the location of the tip. Whilst a tip's position does not inherently encode any information regarding the SIF, as a fracture is invariant with translation in isolation, the fact that no fracture ever truly is in isolation means that its position may become relevant due to the presence of domain boundaries and other fractures. The final feature included for the first iteration of the solution is the angle of the fracture with respect to the tension applied to the domain. This is relevant as the orientation of the fracture with respect to the tension determines which of the SIF elements will dominate, for example if a fracture is perpendicularly aligned to the tension then K_I will be dominant, whereas in parallel alignment K_{II} and K_{III} will become more important.

With the initial feature set identified, a network architecture must be constructed. As there are seven features and three targets, the first and last layers will have seven and three neurons respectively. To keep initial complexity low for proof of concept testing, the first network was then constructed with two hidden layers with 25 neurons each (7:25:25:3). Whilst this was the first network to be implemented, these designs were iterated upon and discussed in detail in the Testing section.

4.4 Design Implementation

The initial implementation of the design was an almost one to one translation of the above into a PyTorch script. The data is presented in the form of a comma separated variable (csv) file, with column headers and an ID column, which was then manually separated into a training and test set. The initial training runs showed that the network could only predict one of the three SIF elements with any degree of accuracy at a time, although even the well predicted element was not very accurate at this stage. The first refinement was to increase the complexity of the network by adding an extra fully connected layer at the end with ten neurons. It was hoped that this would improve prediction under the assumption that the first network was of insufficient complexity to model the system. This assumption was shown to be incorrect, as the new network performed similarly, if not worse, than the first network. The next action was to break the predictor down such that it would only predict one SIF element at a time. This change led to successful prediction of the first element to within 15%

error, however the other two elements were still not effectively predicted. This was expected to be due to lack of data and therefore a larger dataset was constructed. The change to the predictor such that it is trained for only one element at a time has been retained.

With the construction of a larger dataset, a new problem arose. As the data is drawn directly from the simulator, there is the possibility that some invalid values may be included into the dataset. The reasons for a datapoint being invalid vary, but the primary cause is when the tip in consideration for that datapoint has coalesces into another fracture, its SIF data goes to zero. Another consideration is that physically, K_I cannot be negative, and therefore any negative values for such are considered erroneous. Consequently, an ancillary routine was written to clean the dataset prior to training, by simply excising the compromised datapoints. An additional problem that became apparent with the new data is the inability of the network to function with large input values (in the region of a thousand and larger, although even with values in the hundreds performance is decreased). The new dataset contained SIF values in the tens of thousands, which have to be normalised to order tens through division. The consequence of this is that the predicted values returned by the network are orders of magnitude too small, however this is easily corrected by applying the inverse of the division operation applied to the features originally.

The dataset used originally had around 1000 entries; the new dataset had in the region of 50000 entries. With this increase in input data, the accuracy for all SIF elements increased to a point where it was clear that the predictor was functioning. At this juncture, the task that remained was to further increase the prediction accuracy. As before, the first step was to increase network complexity, except unlike before, the new network was massively more complex; a 7:25:100:200:50:1 architecture. This increased the number of trainable weights from 1060 for the second network, not including bias, to 32725, again not including bias. This is an increase of roughly 31 times in the number of weights, and therefore system complexity.

The specific improvements made as a result of these changes are detailed in Section 6.3, suffice to say here that there was a significant improvement in accuracy at the cost of increased training time. However, as discussed prior, training time is not a critical factor as once the network is trained the execution is still very fast in actual usage. Further to this point, increasing the number of epochs for which to train almost universally increases the prediction accuracy, though with diminishing returns beyond a certain point. The reason improvement with epochs is not universal and monotonic is due to increased epochs increasing the liability of the network to overfitting, where the network specialises too much on the training set it is given and loses its capacity for generalisation. This was not seen to be an issue for this particular task however as the training, validation and testing scores were and are all in close agreement, in contrast to an overfit model, which would have a training score noticeably superior to the others. Due to this, regularisation is seen to not be required for this application. Finally, there was significant experimentation at this stage with the specific optimiser, criterion and activation functions. Again, this is discussed in more detail in the Testing section (6.4, 6.5 and 6.6).

The fracture system being analysed has hitherto been a simple, non-interacting system. It is clear that the additional complexity resultant from an interacting system would decrease the

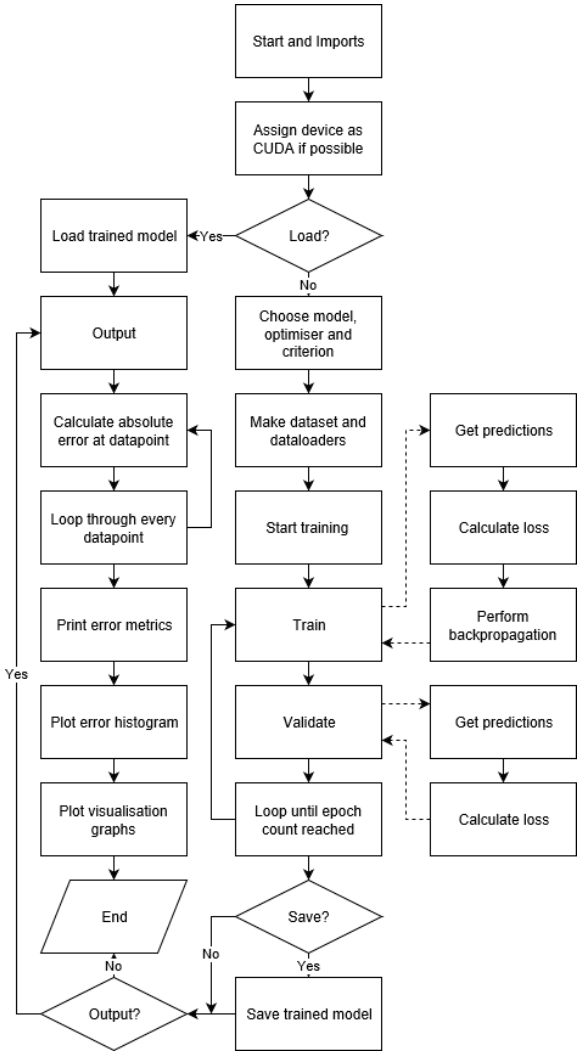


Figure 2: An architectural flowchart showing the operation of the solution. The dotted arrows denote subroutine processes.

accuracy of the predictions, and that is what was seen with the third dataset. The third dataset, once cleaned, numbered around 200000 entries. The data cleaning routine had to be modified slightly for this set as the set contained some truly vast numbers (of the order 10^60), whilst the reasonable physical upper limit for these quantities is 10^8. This was rectified with the addition of an upper limit criterion as well as the existing non-zero requirement. The magnitude of this dataset also led to the final substantial change in the code; instead of manually dividing the set into training and test sets, this division is also performed inside the program, alongside the division of the training set into a validation set. With that change the code has effectively reached its final state.

This stage in the software development lifecycle is where focus is shifted to testing and optimisation, as opposed to direct development. This aspect of development is discussed in detail in its own section.

Whilst the specifics of the code are discussed in Section 5.3, see Figure 2 for an architectural flowchart for the solution.

5 Methodology

5.1 Code Metadata

The solution is written in Python, which was selected due to the existence of powerful Machine Learning libraries written for it. The specific library chosen was PyTorch due to the author's prior experience using it; Keras with Tensorflow is an equally effective alternative.

5.2 Dependencies

As it is written in Python, the code is not compiled, and therefore to run training a Python 3.7 install is required, preferably with Anaconda as it has many of the necessary libraries pre-installed. Further to that, the library dependencies are SciKitLearn, for the explained_variance_score and ShuffleSplit functions; livelossplot for the in-training plotting;

matplotlib for the test plotting; NumPy for generic mathematics; PyTorch as it is core to the entire solution; and pandas in order to read from the data csv files.

However, if all that is required is to run the trained model in a C++ environment, then the only dependencies are LibTorch and CMake.

Whilst not a strict requirement, a CUDA capable GPU will enable the training to occur on the GPU instead of the CPU, leading to improved performance. For this to function Nvidia's CUDA application must be installed and configured.

5.3 Implementation

While the development section prior gives a broad view on the structure of the solution, this section will elucidate upon the finer points of construction.

The solution is written with a strong functional emphasis in order to aid with readability, and to clarify the logical progression of the code. The only elements of the program that exist in the global namespace are the imports, the CUDA assignment and the execution line. CUDA is a service developed by Nvidia to enable a solution to use GPUs as opposed to the CPU, which can lead to massive performance increases with the correct hardware. Consequently, there is an if statement that checks if a CUDA device is available, in which case it will set the torch device object to said CUDA device. Otherwise the torch device will be set as the default CPU.

The first structure in the code is a class for a custom dataset called **TipDataset** that handles both the reading of the data and the presentation of the data to torch later on. It reads the data from a csv file, the address of which is taken as an argument, using pandas. This raw data is held as an attribute called *tips*. The features and targets are then extracted from *tips* in much the same way. First, the *tips* values are turned into a NumPy array, and then the relevant region is sliced out, this being the difference between the features and targets. Then this sliced array is transformed into a torch tensor. Finally, a divisor (can be passed as an argument) is applied to the relevant elements in order to reduce their magnitudes to those which can be used by the network. The features are saved as an attribute called *data*, whereas the targets are saved as an attribute called *lbls* (from labels, an alternative name for targets). The final attribute in the dataset is *ndat*, which holds the number of features in the set. The target must be selected however, and this is achieved by a quick transformation of a passed argument identifying the desired SIF element into the index of said element.

The function responsible for the operation of the entire solution is **main**. The only required argument is the file address of the data file (passed as file_name), however there are many optional arguments that control the behaviour of the program. These include whether to plot the learning curve during training, whether to perform model evaluation after training, whether to save the model upon training completion and whether to load a pretrained model. These flags are plot, test, save and load respectively. In addition, there are arguments to choose which model architecture to use, how many epochs to run training for, which SIF element to predict, to set the unique ID of a model for saving, and the identifier of the model

to be loaded, if loading has been selected. Again, these are the mdl, n_epochs, sif, mID and ld_mdl arguments respectively.

The function then breaks into two halves, depending on whether the load flag is true or false. If false, then a **TipDataset** is constructed for the selected SIF element, and the selected model is initialised and sent to the torch device designated at the start of the program. The optimiser and criterion are then initialised as a torch.optim object and nn object respectively, the data loaders are constructed by **dataldr_make** and the system is ready for training, which is performed by **execute**. Upon completion of training, the model is saved if the save flag is set to true. If the load flag is true, however, then the loading identifier is broken down into its constituent parts, namely the model architecture ID, the SIF element, and trained model ID (set when the model is saved). With these identifiers, the desired pretrained model can be identified and loaded. This operation has several defensive measures including failure flags and a try-except structure to prevent file handling errors. Finally, the **TipDataset** is once again constructed from the provided file address. Regardless of the load flag, the function will then call **output** for evaluation if the test flag is set to true.

As referenced prior, the **dataldr_make** function takes the **TipDataset** and constructs the training, validation and testing dataloaders, as well as returning the test set features and targets. To do this, the ShuffleSplit functionality from ScikitLearn is employed, in order to randomly split the datapoints into their respective sets. As this is a regression problem, not classification, stratified splitting is not required. The initial split is 80/20, where the larger portion becomes the training set, and the lesser portion is then split again. This second split is 50/50, and the products become the validation and testing sets. After the splitting operations, the data exists as separate features and targets for each set, so they are then each concatenated into torch TensorDataset objects, which are then themselves transformed into their respective dataloaders. The dataloaders are then returned, alongside the test features and targets, which are used in model evaluation.

The model architectures are all saved in much the same way, as classes, with their only differences being the construction of said architectures. They are child classes of the nn.Module class, which is the parent class of all neural networks in the torch framework. Their attributes are then the layers in the network as nn.Linear objects, as well as an activation layer again as an nn object. The class contains one non-init method, **forward**, which dictates the order in which the prior defined layers should be applied, returning the value of the final layer as its output (which is always a single, non-activated real number).

The **execute** function actually performs the training cycle, although it is simpler in practice than that description may imply. If the plot flag in **main** is set to true, then a PlotLosses object is initialised. Regardless, there is then a loop for the specified quantity of epochs. For each epoch, **trn**, which handles the backpropagation, is run, followed by **val**, which handles validation. Finally, if plotting, the plot is updated, if not plotting, the current epoch number is printed to allow for the monitoring of progress. At the cessation of training, the now trained model is returned.

The **trn** and **val** functions are very similar, with the only differences being the mode the model is set to, and the absence of backpropagation in the **val** function. Both start by setting the model to the relevant mode (training for **trn**, eval for **val**) and initialising the loss and accuracy variables. They will then iterate over the datapoints in the dataloader. Both the features and target will first be moved to the torch device, then for **trn** the optimiser will be set to zero grad for training. The model will then make its prediction, and the loss subsequently calculated by the criterion, followed by backpropagation if training. The loss variable is then incremented by the loss multiplied by the batch size. The accuracy is calculated using explained variance score (EVS) from ScikitLearn and the accuracy variable incremented, again after multiplication by batch size. In the case of **trn**, the optimiser is then incremented by one step. Once the dataloader has been iterated through to completion, the loss and accuracy are returned, though divided by the length of the dataset to find the average of both.

With the above functions, the solution is fully equipped to train a model. However, a trained model is of little value without evaluation of its performance, which is the purpose of the **output** function. The initial operations are to again set the model to eval, and then to get the model's predictions. Following this, the data is looped over, calculating the absolute error at each point, and also recording the index of the point with the worst error. After the loop, the average error is then calculated. The average absolute value of the targets is also calculated, to put context to the average error. These quantities, as well as the worst datapoint as tracked in the loop, are then printed. Following this, the targets and predictions are sorted with respect to the targets, in order to facilitate graphing. The first output graph is an error histogram. The histogram uses 50 bins in order to provide more resolution on the error distribution. This is followed by a graph of predictions against targets, where points on the y=x line represent a perfect match, and increased distance from this line indicates a worse prediction. The final graph is of the predictions and targets plotted independently, in order to see if certain regions behave in idiosyncratic ways. For example, there is often a vertex in the targets line which the predictions have difficulty following.

There are certain associated utility functions for this solution, used for such purposes as cleaning the dataset and turning a text file into a csv. These functions are of lesser importance and of no static form as they are often edited for specific situations. They will therefore not be discussed.

6 Testing

6.1 Overview

The testing approach is often split into two aspects – validation and optimisation – where validation ensures the code is actually performing the required tasks and producing the "correct" results, and optimisation is concerned with improving the "correctness" of said results once validated.

For this problem, this distinction is hard to draw, as the correctness of a result is even less well defined than normal. Therefore, very few active measures are taken with regard to

solution validation beyond consistent usage of good coding practices and documentation, and the presence of a linter and error detector within the IDE. Furthermore, Continuous Integration (CI) is ill-suited to this problem, again due to the ambiguity regarding the correctness of a given output, and as a result CI was not implemented in the repository. For similar reasons there is no unit testing. However, successful optimisation is predicated on functional code, and therefore the successful optimisation discussed below is taken as evidence that the solution is functioning as expected.

There are a number of factors that require optimisation when implementing an ML system. In this particular case, the primary factors, hereafter hyperparameters, are as follows:

- Feature set
- Model architecture
- Optimiser
- Criterion function
- Activation function
- Number of epochs

There are other tuneable hyperparameters in the system, however those not listed here were left as their default values, so as to keep the optimisation feasible. This is necessary because these factors are all co-dependent, and therefore the optimisation of each in isolation will not necessarily give the optimum configuration. Due to this lack of independence, to find the objective optimum configuration a gradient descent search would have to be performed in as many dimensions as there are hyperparameters which is computationally infeasible. The strategy therefore was to heuristically sort the hyperparameters in order of largest expected effect to smallest, which is the order of the above list, and optimise each in turn (the epoch count does not follow this order, however. This is due to its nature of monotonic increase in accuracy with number epochs until the onset of overfitting. As the degree of overfitting itself is highly dependent on the other hyperparameters, it would be counter-productive to tune the epoch number prior to the conclusion of all other tuning). It is hoped that even though this will not result in the theoretical optimum, it will be relatively close whilst still being a computationally feasible operation.

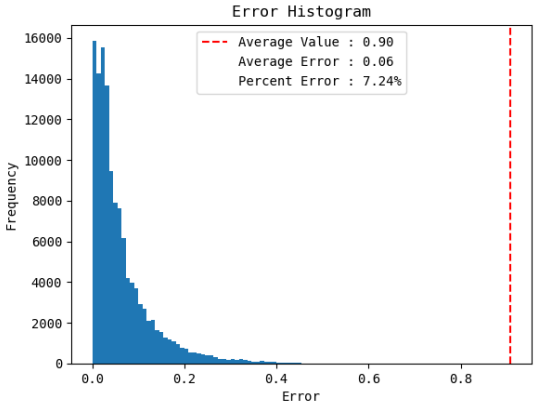
As a further measure to reduce the computational cost, it is assumed during testing that while the accuracies for each of the SIF elements will differ, they will be proportionate. Therefore, the relative accuracies for K_I are taken to be indicative of the relative accuracies of both K_{II} and K_{III} for the purposes of identifying which hyperparameter setting is optimal. This is justified by the relative importance of K_I compared to the other elements, and it is also supported by preliminary testing on this hypothesis.

It should be noted that for all figures presented in this section, the worst 0.5% of errors were omitted from the plots, as these outliers obscure any features of the region of interest. These outliers are still included in the average error calculation however, as their existence is not anomalous and a result of the heuristic nature of neural networks.

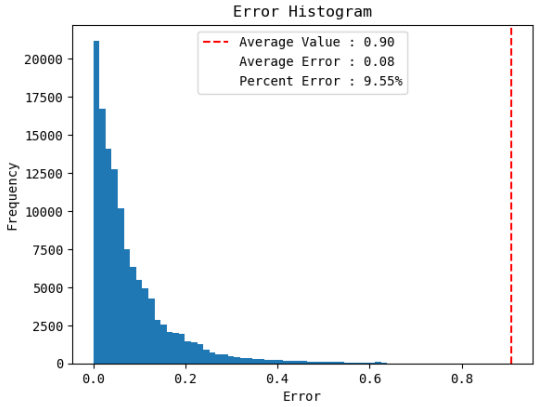
6.2 Feature Set

As mentioned prior, the feature set will often be the most important factor in the performance of a trained model, and is therefore optimised first. The initial feature set consisted of the tip's position in three dimensions, the three current SIF elements, and the tip's orientation to the applied load, the reasoning for which is discussed above in the Design section. In addition to these seven features, two further features were proposed; the normal of the nearest other fracture in three dimensions, and the displacement at the tip. The former was identified with the intention that this information would better inform the model as to the degree of interaction at the tip. The latter is a calculated value, and therefore not of direct utility to the project, but the intention for its inclusion is to see whether the calculated value will significantly improve the predictions. The angle and SIF features will always be retained, however different combinations of the position and orientation features are tested, as well as one simulation with the displacement, such that the input layer may have 7 or 10 neurons.

a



b



c

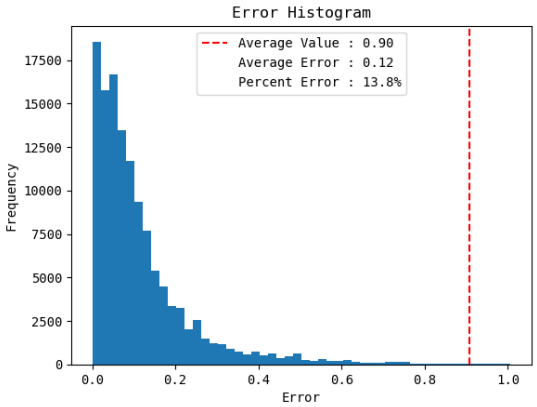


Figure 3
3a: Position, SIF and angle.
This is seen to be the best feature set.

3b: Orientation, SIF and angle.
Inferior to 3a by 2.31%. Expectation that orientation superior to position may be false.

3c: Position, orientation, SIF and angle.
Surprisingly, the combination is worse than either in isolation, implying a degree of unexpected destructive interference.

See Figure 3 for the test results. It is seen that the feature set consisting of the position vector, SIF elements and fracture angle is the optimal selection. This result was unexpected and suggests the position of a fracture may be more important than previously assumed. However, it must be noted that the orientation metric employed was very basic, and therefore may not be the best representation. See Section 7.3 for proposed improvements.

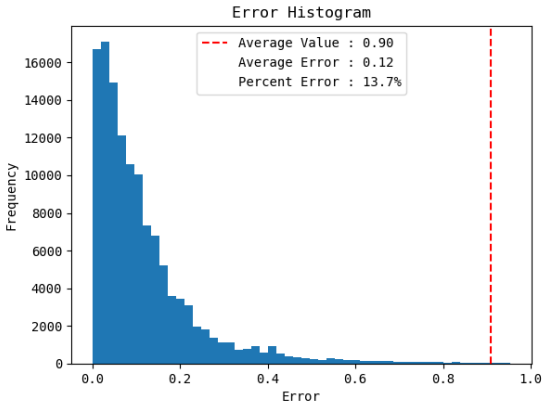
Interestingly, when running the displacement, SIF and angle set, the returned accuracy was 11.5%. This is worse than both position and orientation sets, and only better than the

ineffective combination of the two. This was even more unexpected and warrants further inspection. Although the fact that increased computational time in feature generation does not necessarily have to correlate with superior performance is salient even in isolation.

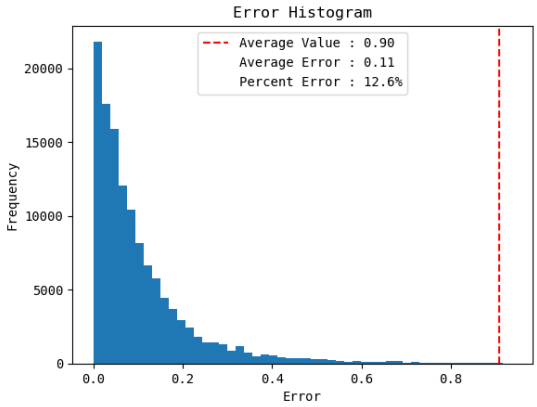
6.3 Model Architecture

The possible design space of a neural network's architecture is infinite and said designs can be arbitrarily complex. Therefore, the design of the network is effectively only bound by the time it takes to train. Despite this lack of design constraint, it shouldn't be necessary for the network to be arbitrarily complex in order to model the system. Consequently, a number of networks were designed in order to identify those which performed best. The actual design process for the network architecture is holistic by nature, as there are no laws describing optimal network construction.

a



b



c

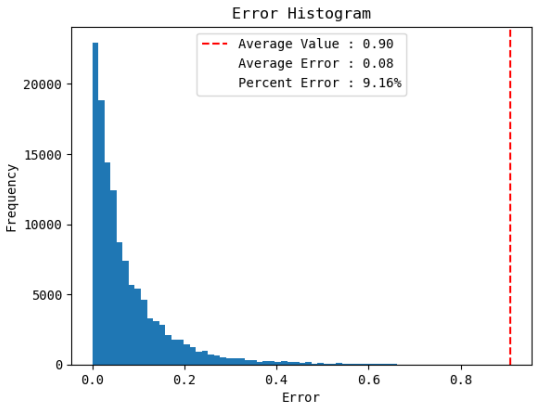


Figure 4
4a: input -> 25 -> 25 -> 1
Unsurprisingly, the most basic network is the worst.

4b: input -> 25 -> 25 -> 10 -> 1
The addition of only a 10 neuron layer leading to a 1% improvement illustrates the learning potential of these networks.

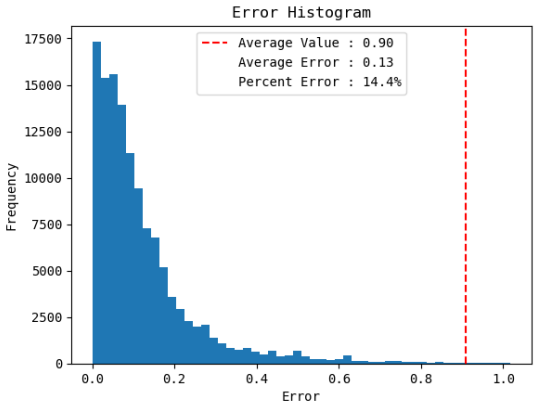
4c: input -> 25 -> 100 -> 200 -> 50 -> 1
The best network by quite some margin.

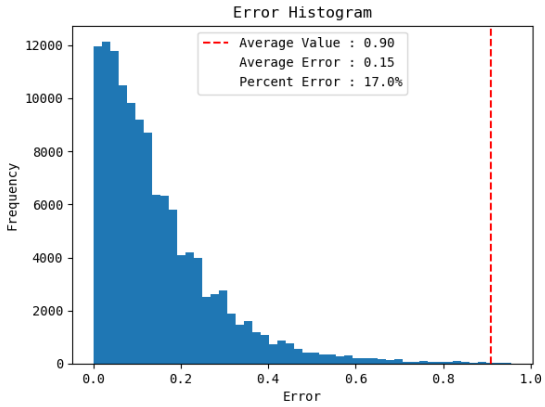
As seen in Figure 4, the more complex network is far superior to the simpler architectures. There is further scope for experimentation in this regard however, and with further development more testing with new architectures would occur at this juncture.

6.4 Optimiser

The optimiser is the function responsible for minimising the loss given by the criterion function. The initial optimiser used was Stochastic Gradient Descent (SGD), however this is a rather simplistic algorithm which just calculates the gradient at a point and descends along this gradient by a certain value (the learning rate). Another tested strategy was RMSprop, which scales the learning rate with the square of the gradient, i.e. it automatically reduces the learning rate near the minimum to combat overshooting. Finally, Adam was also tested, the mechanics of which are too complex for discussion here, but in simple terms it's akin to a combination of RMSprop with non-zero momentum SGD.

a





b

c

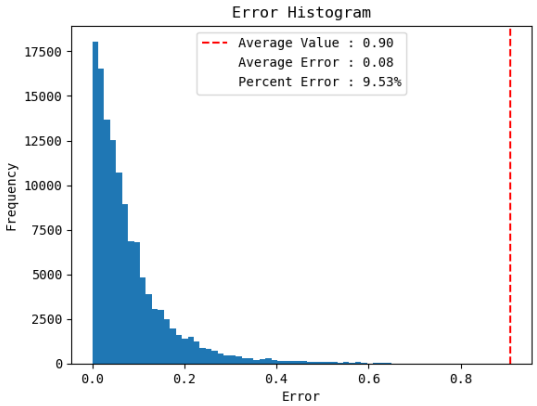


Figure 5
5a: Stochastic Gradient Descent (SGD) Performs well for what is the most basic of optimisers.

5b: RMSprop
RMSprop losing to SGD by 2.6% is not wholly unexpected for relatively smooth gradients.

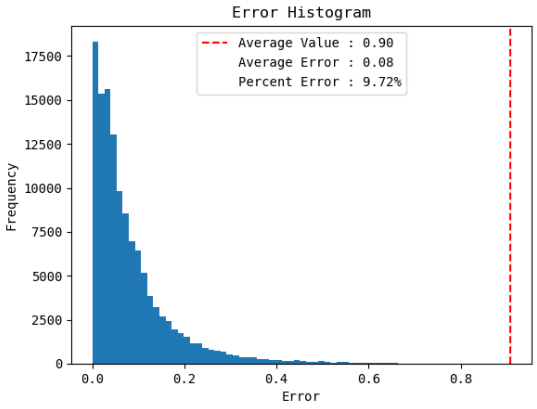
5c: Adam
The most advanced optimiser is superior.

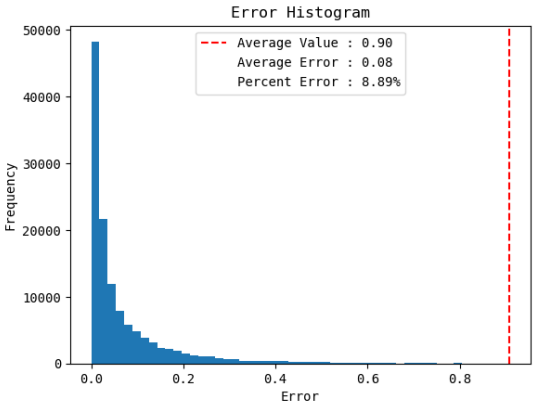
Figure 5 may first appear surprising, as RMSprop performs worse than SGD, however, the principle behind RMSprop is a reduction of learning rate towards the minimum. If the minimum is smooth and shallow, then the more aggressive learning rate of SGD may be of benefit. Although both are inferior to Adam, this is expected as Adam is a very advanced algorithm.

6.5 Criterion Function

The criterion returns a single number that encodes the loss of the system. Three criterion functions were tested: L1Loss, MSELoss (L2Loss) and SmoothL1Loss. L1Loss is simply the mean absolute error, which is resistant to outliers but maintains a large gradient even towards the minimum. MSELoss is the mean squared error, which is effectively the inverse of L1Loss, having an adaptive gradient, but also being vulnerable to outliers. SmoothL1Loss is a combination of the two, where if the element-wise error is below one, it uses MSELoss, and otherwise it uses L1Loss, which ideally results in the best of both algorithms. Whether it is the best will depend on the dataset, however.

a





b

c

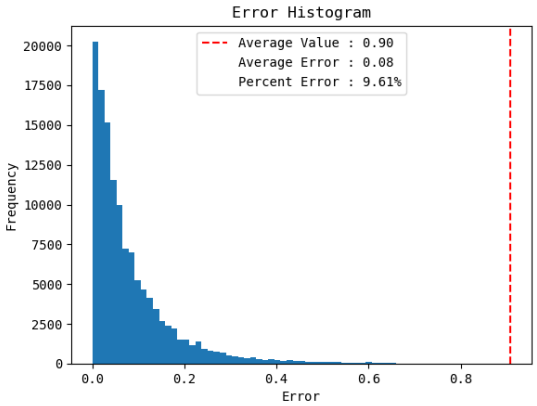


Figure 6
6a: MSELoss
Only 0.11% separates MSELoss and SmoothL1Loss, interesting due to the difference in the criterion curves.

6b: L1Loss
The best function, but all of the tested criteria performed similarly.

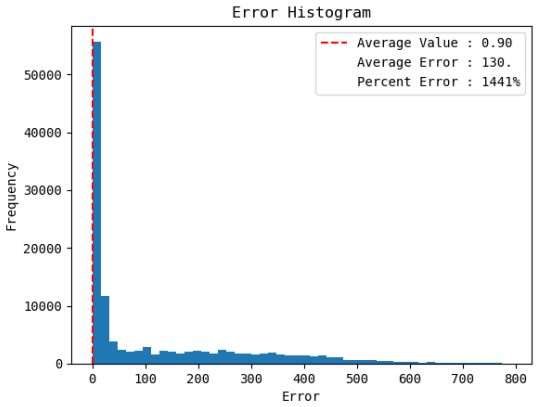
6c: SmoothL1Loss
It is surprising that SmoothL1Loss is inferior to L1Loss, although the difference is not particularly statistically significant.

The test results in Figure 6 are the most divergent from expectation, with all tested criterion functions performing very similarly, and L1Loss, an inferior function upon first inspection, is shown to be the optimal choice. It is also suspected that MSELoss would underperform when predicting K_{II} and K_{III} as their propensity for being negatively valued may adversely affect a function that squares the values, although more testing would be required to ascertain whether this truly is the case. However, as MSELoss is already the worst option, this testing was not pursued.

6.6 Activation Function

The activation function determines whether the output of a neuron constitutes an activation and may also transform said output. The functions tested consisted of Sigmoid, Rectified Linear Unit (ReLU) and LeakyReLU. The sigmoid function is a mathematical function that maps the infinite input domain to an output domain between negative one and one. It has the downside of vanishing gradient away from zero, leading to lesser differentiation between neurons. ReLU is the most popular current activation function, which outputs zero for any negative input and does nothing for a positive input. This does not have the problem of vanishing gradients, however a neuron can become "dead" if its output is always negative, as it cannot be trained with a constant null activation. LeakyReLU attempts to correct this problem by returning a small negative output for negative inputs, instead of outputting zero. The presence of even a small value allows the weights to still be trained.

a



b

c

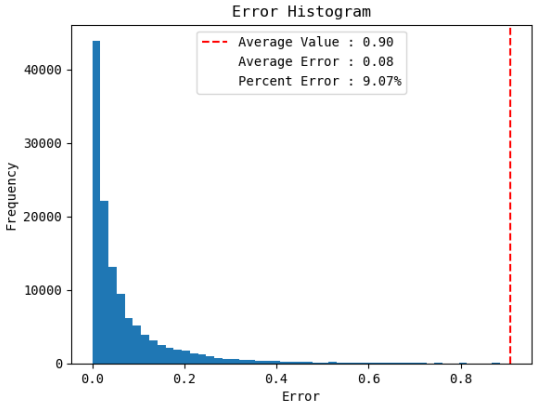


Figure 7

7a: Sigmoid

It is apparent that a sigmoid activation is incapable of learning this system.

7b: ReLU

The poor performance of basic ReLU is most likely explained by the aforementioned dead neuron phenomenon.

7c: LeakyReLU

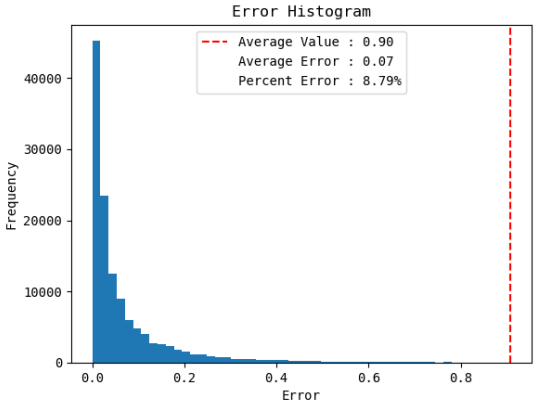
With the rectification of the presumed dead neuron problem, LeakyReLU performs excellently by comparison with the original.

Figure 7 illustrates the unsuitability of the sigmoid function to many complex problems, although the degree to which it failed is remarkable. Whilst LeakyReLU has theoretical advantages over ReLU, the ubiquity of ReLU in the machine learning field makes the 20% difference between the two particularly noteworthy. Further study would be necessary to identify why such a divergence is found, but that is beyond the scope of this project.

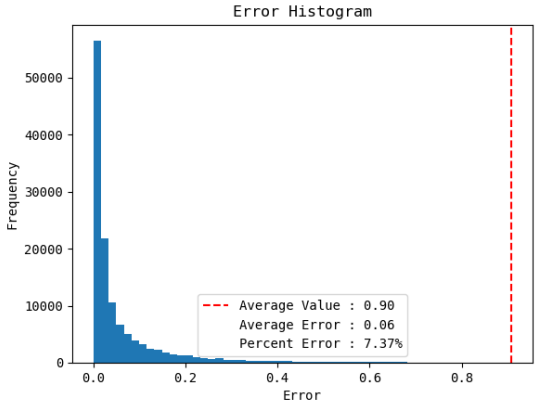
6.7 Number of Epochs

As discussed prior, the reason the number of epochs for which to run training needs to be tuned is to avoid overfitting. Furthermore, while more epochs will always result in more accuracy given the absence of overfitting, there will also be aggressively diminishing returns, so it is important to note the point at which it is unnecessary to continue the training cycle.

a



b



c

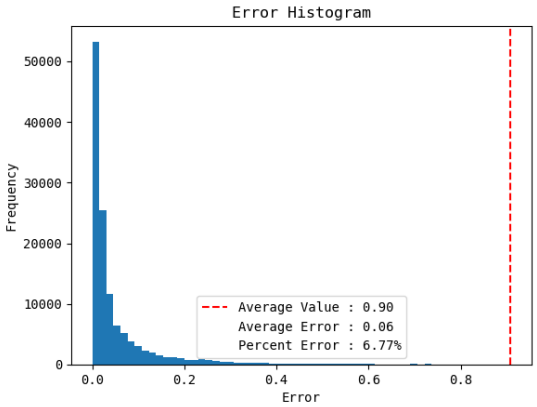


Figure 8
8a: 500 epochs
The expected trend is seen, where the lowest epoch count results in the worst performance.

8b: 1000 epochs
The middle performance, as expected.

8c: 2000 epochs
The lessened degree of improvement from 1000 to 2000 as compared to 500 to 1000 implies the plateau has been reached.

Figure 8 shows what is predicted, though the gradient of improvement's drastic decline shows the expected diminishing returns. The first doubling of epoch count leading to an improvement of 1.42%, followed by the second doubling only garnering 0.6%, suggests that further epochs would not be worthwhile. Observation of the learning curve whilst training supports this conclusion.

6.8 Testing Evaluation

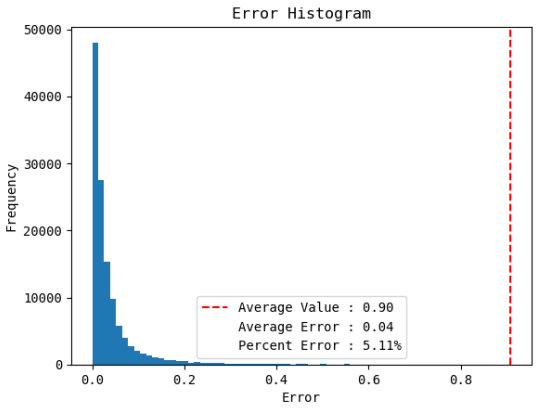
With the cessation of testing, the following configuration is determined to be optimal with regard to the results gathered; a feature set of position, SIF and angle; the third network architecture of input:25:100:200:50:1; an Adam optimiser; L1Loss criterion and LeakyReLU activation function. The epoch count for the final runs is set to 1000 to enable faster development. Despite this, further developmental testing was employed with 2000 epoch runs and even a 5000 epoch run, which unfortunately corrupted during saving. The learning curves observed during training did however strongly suggest no significant benefit would be garnered from greater epoch counts.

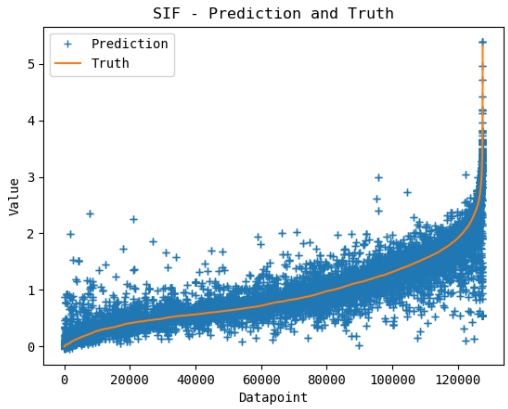
7 Discussion and Conclusions

7.1 Results and Discussion

Following the testing and optimisation program detailed in the previous section, a final run was performed using the configuration established, except unlike during testing, a predictor was trained for all three of the SIF components. The dataset used is the same as that used for the testing, namely an interacting system.

a





b

c

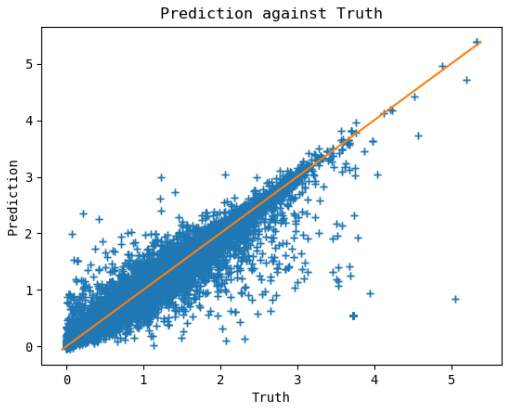


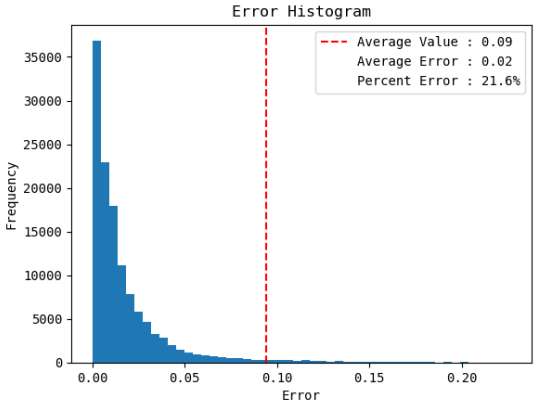
Figure 9: K_I
9a: The error here is superior to that achieved by 2000 epochs in Figure 8c. This further validates the testing strategy, as an optimised configuration outperforms a weaker configuration run for twice the epochs.

9b: The predictor can be seen to follow the sharp upward trajectory that the data takes, although possibly at the loss of accuracy on the flatter region.

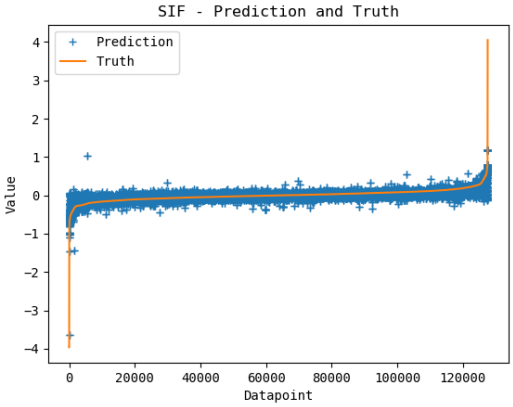
9c: This expounds on 9b, showing that while the predictor can follow the larger values, it also predicts far smaller values occasionally.

The results for K_I are shown in Figure 9. In 9b, one can see that relatively very few targets have values more than double that of the main body of the data. It would appear that despite accurate prediction for these particular targets, the standard values overall seem to undershoot. Furthermore, while the overall distribution follows the true values quite accurately, at the vertex where the targets rapidly climb in magnitude, there are many gross underestimates as well. From this one can posit that the network has difficulty with rapid changes in gradient, which will be emphasised with the remaining SIF components.

a



b



c

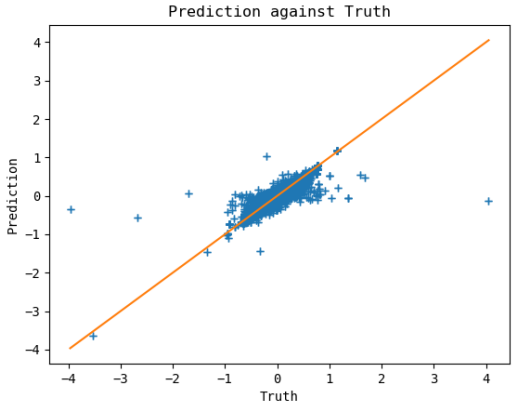
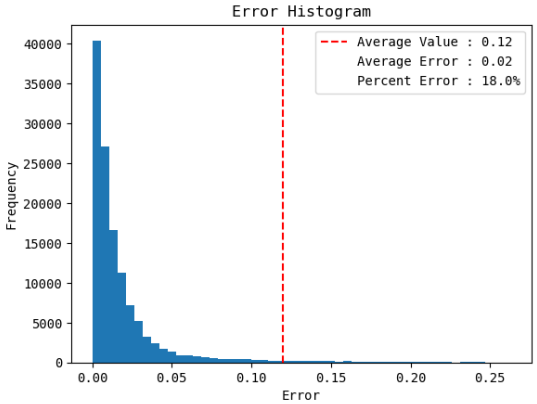


Figure 10: K_{II}
10a: There is a significant rise in error, due to the increased complexity of mode II damage.

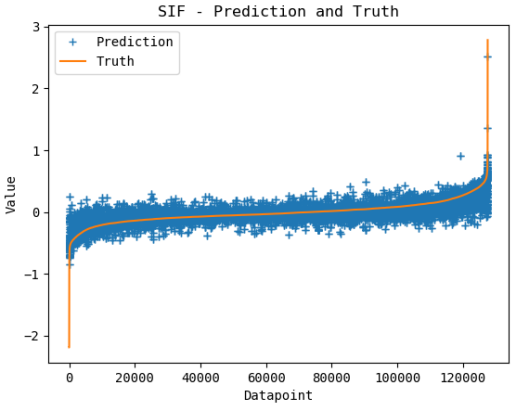
10b: It is seen that the predictor struggles with following the tails of the distribution.

10c: The difficulty seen in 10b is explained here; there are only a handful of points with magnitude greater than 1. This is not enough data for effective learning.

a



b



c

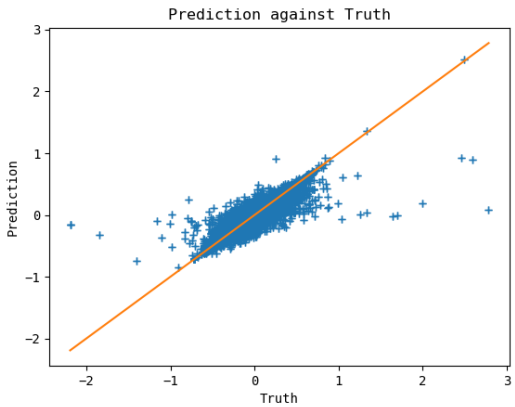


Figure 11: K_{III}
11a: Whilst marginally better than K_{II}, K_{III} shows similar behaviour for similar reasons.

11b: The improvement over that seen in Figure 10 may be explained by the reduced magnitude of the outliers.

11c: One can see that though the outliers do remain, there is a tighter domain with more outlying points. This concentration of data aids in the training process.

Figures 10 and 11 for K_{II} and K_{III} respectively illustrate the added complexity of shearing damage compared to opening. There are a number of reasons as to why this adversely affects the predictor's accuracy. Firstly, there is significantly less variation between the datapoints; while the bulk of the data for K_I spanned from 0 to 2, for the shearing components this was reduced to -0.5 to 0.5 in both cases. The reduction in variation obscures any possible trends, patterns or relationships that the predictor is trying to learn.

Secondly, while K_I is positive definite, K_{II} and K_{III} are equally likely to be either side of zero, as shearing can occur in either direction, while negative opening is impossible in this framework. The corollary to this fact is that the magnitudes of K_{II} and K_{III} are reduced as a result of their probability space being centred on zero. This compounds with the lack of variance discussed above to further obfuscate any patterns, which may otherwise be apparent if the system was scaled up.

Finally, Figures 10c and 11c show how compact the domain is in comparison to that seen in Figure 9c. The lack of data density at the periphery of the dataset prohibits the network from learning how to predict values in these regions, whilst their presence nonetheless influences the loss function. This influence may well be disproportionately large, as the loss function punishes larger errors more harshly, which incentivises the network to train for large but infrequent datapoints.

The exclusion of such outliers is an attractive solution, however their presence in the dataset is a direct consequence of such values existing in the simulator, and therefore they must remain as part of the dataset if this solution is to be incorporated into said simulator.

7.2 Conclusion

The results presented show the viability of a Machine Learning approach to the acceleration of a fracture simulator by exhibiting the clear capability to follow distribution of the target data. It is also apparent that K_I is far easier to predict than K_{II} and K_{III} as is expected, and therefore the application of this strategy to systems with more oblique loads liable to generate shearing must be carefully monitored, whereas a more orthogonally aligned configuration would be quite amenable to the acceleration strategy developed herein.

Furthermore, as almost all practical applications of fracture simulations require interacting systems, while these results show the capacity for this approach to be an effective predictor for these systems, the current accuracy is not sufficient for practical usage. Strategies for engaging with fracture interactions on a deeper level are discussed below, with the intention that the addition of features that encode fracture interaction parameters would increase the network's capacity to learn these systems, and therefore achieve accuracy superior to that which is currently seen. Simple expansion of the dataset would also almost certainly lead to an improvement in system accuracy, as well as running for more epochs, despite the aforementioned inefficiency of such an approach to performance improvement.

However, when taken as a proof of concept for the use of ML strategies for fracture simulation acceleration, the project can be seen as a success and a foundation for further development and iteration on the idea.

7.3 Future Developments

All ML systems can be improved with further refinement of hyperparameters, and this project is no exception. The hyperparameter tuning performed for this solution was somewhat restricted due to time pressure. With further time, a hyperparameter tuning framework would be constructed to replace the current tuning regime. This framework would begin by performing a random series of configuration tests (as random testing is superior to a grid search due to its ability to cover more distinct possibilities as opposed to remaining within a small vicinity) to gain a base knowledge of the hyperparameters. The framework would then perform a gradient descent in order to find an optimal configuration. This would allow more hyperparameters to be tuned, such as the batch size for training and testing and the learning rate and momentum, among others. The issue with this strategy is the prohibitive time cost required to perform a gradient descent in so many degrees of freedom, however now that the codebase is mature, more time could be dedicated to this aspect of the project.

There are also a number of possible minor alterations to the code that may increase performance and usability. An example would be automatic termination of training if the learning curve has plateaued, or automatic detection and preparation of csv contents, as currently these operations must be performed externally to the main body of the solution.

A more advanced addition would be the inclusion of inter-fracture geometric data into the feature set, beyond the currently implemented normal orientation, which was shown to be ineffective (Section 6.2). The lack of data of this kind explains the fall in accuracy in multi-fracture systems by comparison to lone fracture systems, as fractures can have great effect on one another. Unfortunately, the nature of this interaction is not as simple as the distance between two fractures, as the orientation is also a large factor in the interaction. The author proposes a metric that would concatenate as much of this geometry into a single number, in order to use as a feature. An example metric would be

$$\Lambda = \frac{\theta}{s} \cdot \left(\frac{\pi}{4} - \varphi\right)$$

where

$$\theta = \left| |\theta_1 - \theta_2| - \frac{\pi}{2} \right|$$

and s is the distance between the two fractures, φ is the complement of the angle between the major axes of the fractures, θ_1 and θ_2 are the angles of the major axes of the fractures from the perpendicular of the applied load, and Λ is the metric. See Figure 12 for a schematic representation.

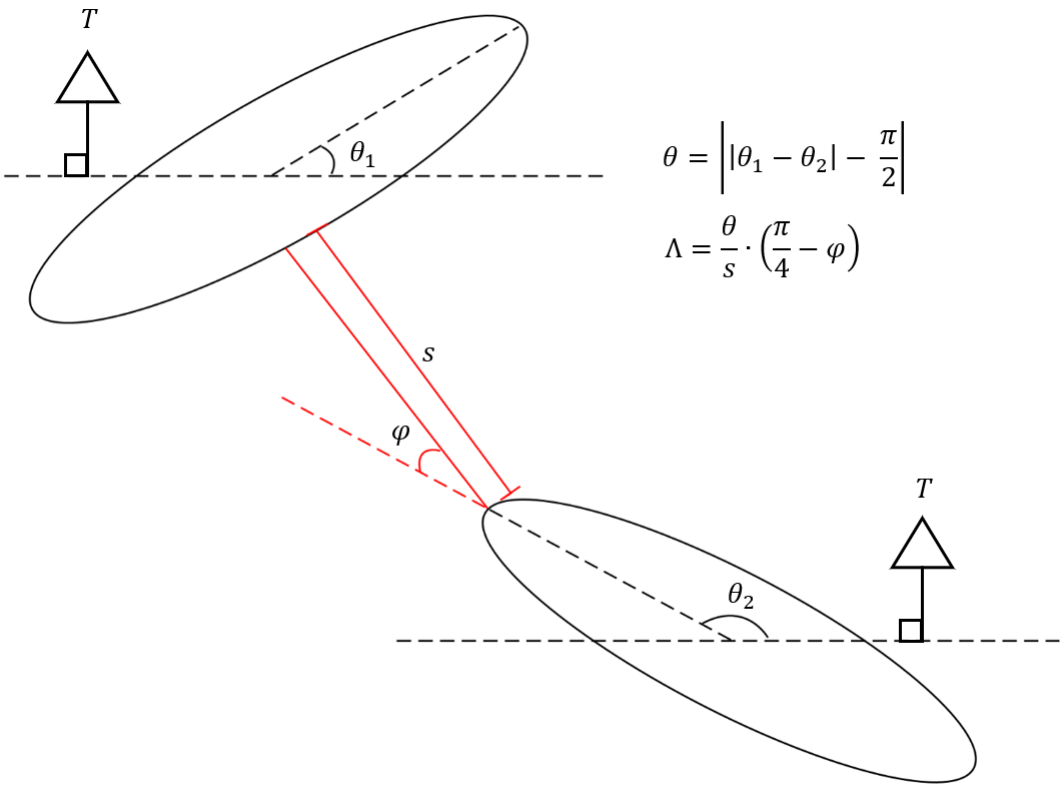


Figure 12: A schematic representation of the proposed orientation metric. T denotes the direction of the applied load.

There is also the obvious extension of implementing the second proposal, which was not attempted due to time constraints. The implementation of the second proposal could also be combined with the current solution, in order to inform the simulator as to whether a fracture tip can be accurately predicted by the network, or whether it has too high of a degree of interaction with other fractures for this prediction to be valid.

The synthesis of this combination with the metric described above would yield a predictor that can use mechanical and geometric data to predict the SIF, and also to know when this prediction is trustworthy or not. It is expected that such a predictor would be superior to that which has been presented herein.

8 Acknowledgements

I would like to thank my supervisor, Dr Adriana Paluszny Rodriguez, for her continual and unerring support and guidance, without which this project would not have been possible. I would also like to thank the extended Rock Mechanics Group in the department for the usage of their Imperial College Geomechanics Toolkit simulator, which formed the basis for the training data. Finally, I would like to thank my colleagues on the Applied Computational Science and Engineering course for their illuminating discussions and encouragement.

9 References

[1] Knight, E. E., Rougier, E., Lei, Z. & Munjiza, A., 2014. *Hybrid Optimization Software Suite,* s.l.: US DoE, OSTI.

[2] Paluszny, A., Matthai, S. K. & Hohmeyer, M., 2007. Hybrid finite element–finite volume discretization of complex geologic structures and a new simulation workflow demonstrated on fractured rocks. *Geofluids,* Issue 7, pp. 186-208.

[3] Paluszny, A. & Zimmerman, R. W., 2011. Numerical simulation of multiple 3D fracture propagation using arbitrary meshes. *Comput. Methods Appl. Mech. Engrg.,* Issue 200, pp. 953-966.

[4] Paluszny, A., Tang, X. H. & Zimmerman, R. W., 2013. Fracture and impulse based finite-discrete element modeling of fragmentation. *Comput Mech,* Issue 52, pp. 1071-1084.

[5] Paluszny, A., Salimzadeh, S. & Zimmerman, R. W., 2018. Finite-Element Modeling of the Growth and Interaction of Hydraulic Fractures in Poroelastic Rock Formations. In: *Hydraulic Fracture Modeling.* s.l.:Elsevier, pp. 1-19.

[6] Moore, B. A. et al., 2018. Predictive modeling of dynamic fracture growth in brittle materials with machine learning. *Computational Materials Science,* Issue 148, pp. 46-53.

[7] Hunter, A. et al., 2019. Reduced-order modeling through machine learning and graph-theoretic approaches for brittle fracture applications. *Computational Materials Science,* Issue 157, pp. 87-98.

[8] Cybenko, G., 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems,* 2(4), pp. 303-314.

10 Bibliography

Valera, M. et al., 2018. Machine learning for graph-based representations of three-dimensional discrete fracture networks. *Computational Geosciences,* Issue 22, pp. 695-710.

Schwarzer, M. et al., 2019. Learning to fail: Predicting fracture evolution in brittle material models using recurrent graph convolutional neural networks. *Computational Materials Science,* Issue 162, pp. 322-332.

Schmidhuber, J., 2015. Deep learning in neural networks: An overview. *Neural Networks,* Issue 61, pp. 85-117.

Breiman, L., 2001. Random Forests. *Machine Learning,* Issue 45, pp. 5-32.

Fukushima, K., 1980. Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position. *Biological Cybernetics,* Issue 36, pp. 193-202.