INL EXT-21-63619

]

Structural Health Monitoring of Microreactor Safety Systems Using Convolutional Neural Networks

July 2021

Erin Yan,
Carnegie Mellon University

Piyush Sabharwall,
Idaho National Laboratory

Harleen Kaur Sandhu, Saran Srikanth Bodda, Abhinav Gupta, and Xu Wu
North Carolina State University

Idaho National Laboratory

INL is a U.S. Department of Energy National Laboratory operated by Battelle Energy Alliance, LLC

DISCLAIMER

This information was prepared as an account of work sponsored by an agency of the U.S. Government. Neither the U.S. Government nor any agency thereof, nor any of their employees, makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness, of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the U.S. Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the U.S. Government or any agency thereof.

INL EXT-21-63619

Structural Health Monitoring of Microreactor Safety Systems Using Convolutional Neural Networks

Erin Yan,
Carnegie Mellon University
Piyush Sabharwall,
Idaho National Laboratory
Harleen Kaur Sandhu, Saran Srikanth Bodda, Abhinav Gupta, and Xu Wu
North Carolina State University

July 2021

Idaho National Laboratory
Originating Organization [optional]
Idaho Falls, Idaho 83415

http://www.inl.gov

Prepared for the
U.S. Department of Energy
Office of _______
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

Page intentionally left blank

ABSTRACT

Microreactors are a class of modular reactors, which have innovative applications in nuclear and nonnuclear industries due to their portability, reliability, resilience, and high capacity factors. In order to operate microreactors on a wider scale, it is essential to bring down maintenance life-cycle costs while ensuring the integrity of operating such systems. Autonomous operations in microreactors using augmented digital-twin (DT) technology can serve as a cost-effective solution by increasing awareness about the system's health. Structural health monitoring (SHM) is a key component of nuclear DT frameworks. Artificial neural networks can be beneficial to detect degradation in the nuclear safety systems, such as piping equipment systems, by monitoring the sensor data obtained from the plant and its corresponding structures, systems and components. In this report, an SHM methodology is presented which uses convolutional neural networks to determine degraded locations and their corresponding degradation-severity levels at various locations of nuclear piping equipment systems. A simple pipe system, subjected to seismic loads, is selected to design the post-hazard SHM framework. The effectiveness of the proposed SHM methodology is demonstrated by obtaining high accuracy in detecting degraded locations as well as the severity levels.

Page intentionally left blank

CONTENTS

ABSTRACT iii

ACRONYMS viii

1. INTRODUCTION 10
   - 1.1 OVERVIEW OF ARTIFICIAL NEURAL NETWORKS 11
   - 1.2 Feed Forward Networks 11
   - 1.3 Recurrent Neural Network 12
   - 1.4 Convolutional Neural Networks 13
2. PROJECT DESCRIPTION 13
3. METHODOLOGY 14
   - 3.1 Data Preprocessing 14
   - 3.2 Creating the Convoluted Neural Network 15
   - 3.3 RESULTS 16
4. FUTURE WORK 17
5. ACKNOWLEDGEMENTS 18
6. REFERENCES 18

Appendix A Creatiing the CNN 21

FIGURES

Figure 1. Single neuron. 11

Figure 2. Feed-forward network (FFN). 12

Figure 3. RNN. 12

Figure 4. CNN. 13

Figure 5. Simple piping system. 14

Figure 6. STFT surface plot of each direction of each sensor. 15

Figure 7. CNN architecture. 15

Figure 8. Sensor optimization. 17

Figure 9. Importing analytic programs and mounting Google Drive. 21

Figure 10. Standardizing random shuffling of the data. 21

Figure 11. Data iteration. 22

Figure 12. Data flattening. 22

Figure 13. Data normalization. 22

Figure 14. Separating data between training and testing groups. 23

Figure 15. Setting parameters and hyperparameters. 23

Figure 16. Creating the CNN. 24

Figure 17. Running the model. 25
Figure 18. Final data run. 25

TABLES

Table 1. ANN training parameters. 16
Table 2. Accuracy of proposed SHM framework. 17

Page intentionally left blank

ACRONYMS

AI	artificial intelligence
ANN	artificial neural networks
CNN	convolutional neural networks
DT	digital twin
FFN	feed-forward network
GPU	graphics-processing unit
GRU	gated recurrent unit
LOCA	loss of coolant accident
LSTM	long short-term memory
NAMAC	Nearly Autonomous Management and Control
NDT	non-destructive testing
OM	operation and manufacturing
RNN	recurrent neural networks
SHM	structural health monitoring
STFT	short-time Fourier transform
TPU	tensor-processing unit
USNRC	United States Nuclear Regulatory Commission

Page intentionally left blank

Structural Health Monitoring of Microreactor Safety Systems Using Convolutional Neural Networks

1. INTRODUCTION

Microreactors have the potential to provide factory-fabricated, safe, and transportable nuclear energy to civilian, industrial, and defense industries. Despite their relatively small physical footprint, they can provide up reliable and long-lasting clean energy that will not require refueling for years. These novel features open new potential for nuclear-reactor applications in traditionally non-nuclear markets. However, due to the smaller energy output levels of microreactors, operation and manufacturing (OM) costs could be significant due to economies of scale when compared to larger units. These restrictions limit the possibility of having as many supervisory personnel as traditional reactors. Reactor autonomy could enable minimized operations staffing that could allow for cost competitiveness, shorter response times, and newer reactor concepts. Thus, in order for microreactors to become more widespread while maintaining the integrity of safe operations, it is imperative that some level of autonomy be implemented.

With advances in control algorithms and artificial intelligence, autonomy in microreactors by utilizing the digital-twin (DT) concept [1–7] looks achievable. In reactor autonomy, there are various degrees of automation, ranging from providing action alternatives to decidingand acting independently without an operator. Choosing a level of autonomy depends on the tradeoff between staffing, operational flexibility, system flexibility, and safety. A highly autonomous system should demonstrate reliability with little human assistance and be able to process all operating modes by planning actions based on sensor data and identifying the subsequent consequences of its actions.

Artificial neural networks (ANNs) can be used to develop an autonomous reactor-control system. Structural health monitoring (SHM) is an important component of the DT framework for any nuclear power plant (NPP) [8–11]. In the proposed research, an SHM methodology is developed for the safety systems of nuclear reactors, such as the piping equipment systems that carry coolants (such as water, sodium, gas, or molten salt) to the reactor vessel and steam to the turbine. A compromise on the structural integrity of piping equipment systems, like a crack or fissure, can result in nuclear accidents such as a loss of coolant accident (LOCA). Thus, maintaining the operational functionality of such safety systems is required. However, the current non-destructive testing (NDT) techniques applied on nuclear piping equipment systems can be time and cost intensive. An SHM framework with the ability to detect locations with all levels of degradation—minor, moderate, or severe—could act as an additional aid to current NDT techniques. It could also prove to be economical by collecting sensor data from the system after an external-hazard scenario, such as an earthquake, and detecting degraded locations to aid NDT procedures, thereby reducing reactor outage periods.

The proposed research focuses on developing an SHM framework for detecting degradation in nuclear safety systems, such as piping equipment systems, by employing state-of-the-art artificial intelligence (AI) algorithms on acquired sensor data [10, 11]. A proof of concept using convolutional neural networks (CNNs) is explored by collecting sensor data from high-fidelity synthetic simulations for a simple piping system subjected to seismic hazards. Processing large amounts of data and extracting degradation-sensitive features remains a fundamental challenge of any robust SHM methodology. In this study, signal processing and feature extraction are carried out on the acquired sensor data in order to create a data repository of degradation-sensitive quantities used to train the CNN algorithms. Effectiveness of a sensor-placement strategy is also investigated. The efficiency of the proposed AI SHM framework is demonstrated for a post-hazard scenario in nuclear piping-equipment systems.

2. OVERVIEW OF ARTIFICIAL NEURAL NETWORKS

Deep learning through artificial neural networks can be applied in many applications that are hard to traditionally program such as imaging classification, speech recognition, natural language processing, and object tracking. They are able to model non-linear and complex problems as well as predict based on unseen data. Inspired by the biological neural networks, artificial neural networks (ANNs) are made of a group of connected artificial neurons (see Figure 1). Each neuron takes in a signal and, after processing, returns another signal to connecting neurons. In each node and connection (edge) there is a weight that is adjusted throughout the learning. Edges are neurons, usually organized in layers, that are interconnected. Multilayer ANNs use back-propagation where the outputs are compared with the correct answer and the difference is put through a loss function. Then the error is fed back through the network so that the model can learn from it.

Not all problems can be solved in a linear fashion, so activation functions must be used to capture non-linear relationships. An activation function also makes the input more useful for the next layer. For example, the most-popular activation function, reLU, simply returns the input value if it is greater than 0, and it returns 0 if the input is less than 0. This makes it easier and faster for the model to handle.

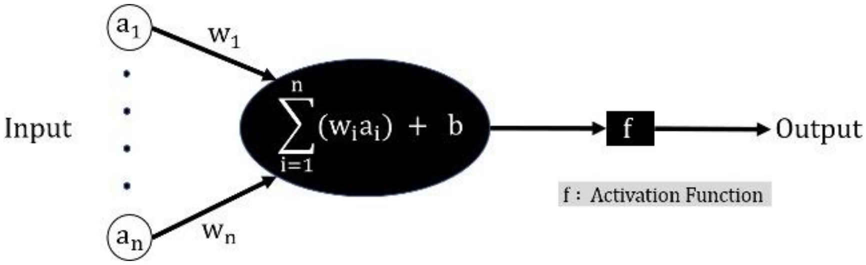


Figure 1. Single neuron.

2.1 Feed-Forward Networks

The simplest way to organize layers of neurons is in feed-forward layers, meaning data only moves in the forward direction. The first layer takes in the input, and the last layer produces outputs. The middle layers are called hidden layers because they are not connected to the outside world. Each neuron in the hidden layer gets the output of all neurons in the previous layer, and if the sum is greater than a threshold, the neuron returns 1. Otherwise, it returns -1. The hidden layer functions as a distiller and finds important features from the input and passes them to the next layer.

When training an ANN, the network first does a forward pass and makes a prediction. The prediction is then compared to the correct result through a specified loss function and the error value is back-propagated through the network to find the gradient. Gradients are how much a network needs to adjust in one layer, but gradients for future layers depend on the gradient of the past layer (see Figure 2). This leads to a diminishing gradient which can lead to minimal learning.

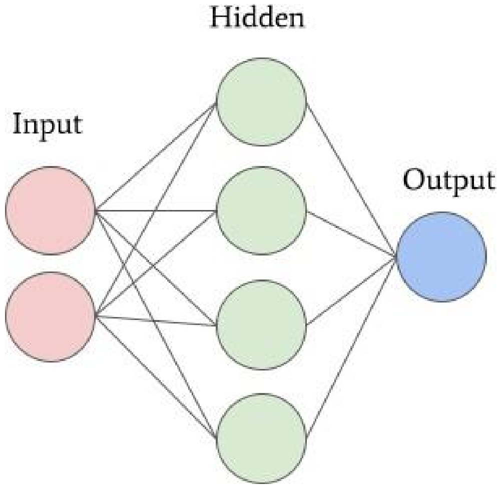


Figure 2. Feed-forward network (FFN).

2.2 Recurrent Neural Network

Recurrent neural networks (RNNs) are another common class of ANNs that use time-series or sequential data. Instead of keeping inputs and outputs separated, as do FFNs, RNNs assume that the input and outputs are dependent. Outputs from previous inputs are called hidden states and are considered when calculating the outputs of future inputs. However, as an RNN processes more steps, it suffers from short-term memory because it contains data of all previous outputs (see Figure 3). As with FFNs, back-propagation leads to vanishing gradients in RNNs. As the network goes through each time stamp, the gradient will shrink exponentially. Smaller gradients lead to smaller adjustments in earlier layers, meaning they will not learn. Due to this issue, long short-term memory (LSTM) and gated recurrent unit (GRU) were created. By using gates, these models can minimize short-term memory through adding and removing from the hidden state. Despite its drawback from short-term memory, RNNs are an excellent choice for neural networks due to their consideration of past information and consistent model size, even with additional input. RNNs are not able to predict labeled data or classify data without their output being processed by a different type of ANN. Hence, in this study, we have not investigatedRNNs.

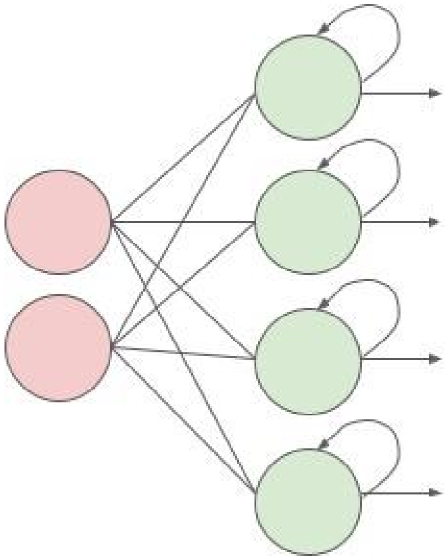


Figure 3. RNN.

2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a class of deep neural networks that can be used for image classification requiring little data preprocessing. They can be used over a large amount of sensor data without requiring prior knowledge or human interference in feature mapping. Like other FFNs, CNNs contain an input layer, at least one hidden layer, and an output layer. The input layer usually is a tensor that has a certain shape. The convoluted layer uses convolution-kernel slides across the input and generates a feature map for future layers. Filter sizes and strides can be modified to optimize the model. The convoluted layer is customarily used adjacent to other layers, such as pooling normalization layers and fully connected layers. Pooling layers take the maximum or average of the current view to prevent overfitting the data. Fully connected layers are usually used at the end of the model to optimize objectives (see Figure 4). In this research, CNN architectures and key parameters, sensor placement strategy and various data-handling techniques are investigated as a part of the proposed SHM methodology.

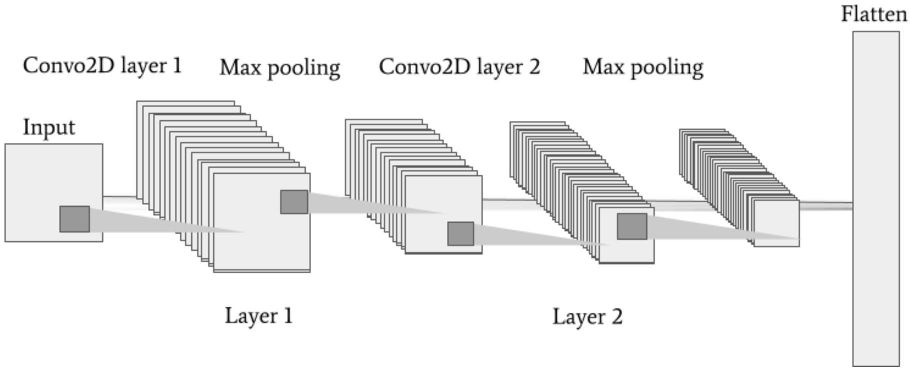


Figure 4. CNN.

3. PROJECT DESCRIPTION

Many components, along with information about their current state, can govern the operational functionality of nuclear facilities. One such component is the distributed piping equipment safety system, which connects one equipment, such as the reactor, a pump, steam generator, to another. When nuclear reactors have been operational for some time period, their piping equipment systems can experience pipe-wall thinning and subsequent reduction in structural stiffness due to phenomena such as flow-assisted erosion and corrosion [12]. Thus, degradation in these systems can be characterized as a reduction in the cross-sectional area of piping elements. Usually, discontinuities, such as elbows, t-joints, and nozzles, in the system experience greater degradation from pipe-wall thinning [13–18]. An SHM framework can prove to be beneficial in extracting information about the system's current, degraded state. This can result in an increased level of monitoring, control, supervision, and security by allowing the operators to obtain knowledge about the degradation severity at various locations of a safety system instead of relying on a predetermined maintenance schedule.

Machine-learning algorithms, specifically ANNs, can be used to design a robust SHM framework and to detect the condition of reactor subsystems. Large amounts of sensor data can be acquired from such nuclear safety systems. As a subset of ANNs, CNNs can be employed as a powerful tool to process, store, and train on this sensor data [19]. Degradation is characterized as minor, moderate, or severe. To develop a post-hazard health-monitoring framework that detects degraded locations as well as the severity of degradation, a simple pipe system, subjected to a collection of 100 earthquakes, is selected as the case study. The simple piping equipment system selected for the first case study is quite similar, but not identical, to a United States Nuclear Regulatory Commission (USNRC) piping benchmark problem [20].

4. METHODOLOGY

4.1 Data Preprocessing

Vibration-based monitoring is used for a variety of civil-engineering applications to detect changes that may indicate damage or degradation. A finite element model of the simple piping system was created using ANSYS FE software. Seismic loads were applied to the model, and high-fidelity simulations were carried out to acquire sensor data in the form of acceleration-time-series signals to capture the phenomenon of degradation in nuclear piping equipment systems. The sensor response acquired from simulation contains over 30 Gb of data storage. A total of nine sensor locations (as shown in Figure 5) are considered and degradation is assumed to occur at one location at any given time.

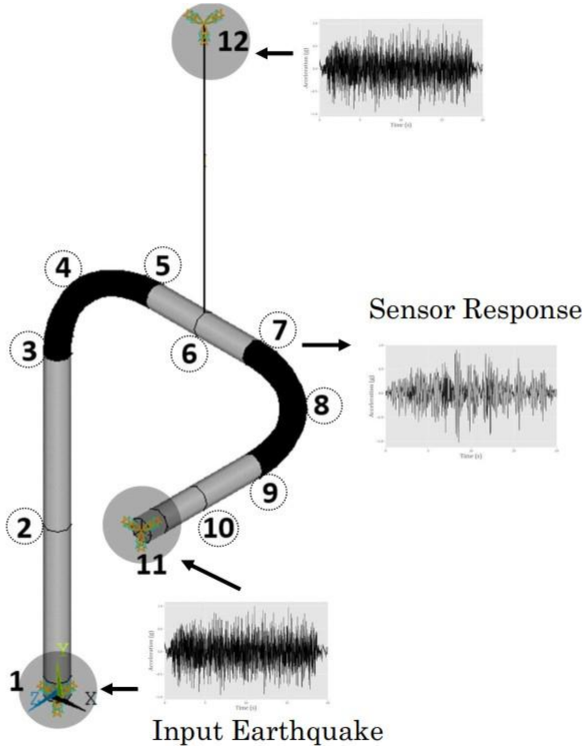


Figure 5. Simple piping system.

Due to the large amount of data acquired, a computationally effective data-processing and storage technique had to be implemented to store the acceleration-time-series data. Using the SciPy [21] short-time Fourier transform (STFT) [22] module, the time series data of each direction of each sensor was transformed into a three-dimensional (3D) array with frequency, time, and STFT magnitude as its axis. Then the plots were cropped at 100 hz because the STFT magnitude after 100 hz was mostly zero. For each simulation, the STFT surface plots were consolidated into a Pickle file [23] so they could be compactly stored in a machine-learning data repository. Pickle storage was chosen over Tensorflow Records [24] due to its faster saving and loading time in this particular case study (see Figure 6).

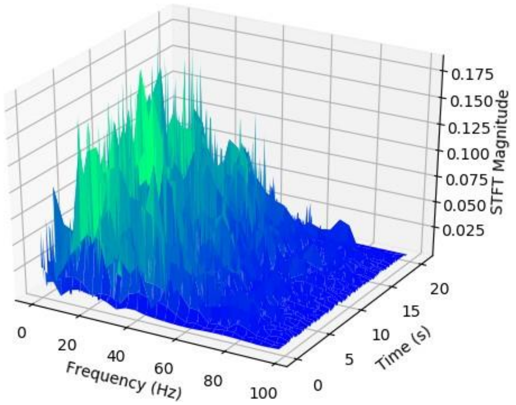


Figure 6. STFT surface plot of each direction of each sensor.

4.2 Creating the Convoluted Neural Network

From previous literature [19], it is known that a CNN created using Python's Keras library [25] can be used to analyze STFT plots. In this, a similar structure was used, but with three convolutional layers instead of four due to the smaller size of our plots. Because a max pooling layer sits between every layer to enhance feature extraction, image resolution decreases after each layer. This limits the possible depth of the model. A dropout, which is a percentage of neurons that are randomly ignored is also implemented after each layer to prevent overfitting. The two-dimensional (2D) CNN model (see Figure 7) is constructed to be trained over 100 epochs using the STFT data for image classification. The rectified linear activation function (reLU) was selected as the activation function for each layer due to its speed and ability to overcome the vanishing-gradient problem. Softmax was then selected as the output-layer activation due to its performance on multiclass classification problems. An adaptive learning rate was implemented so that the learning rate would decrease on plateau.

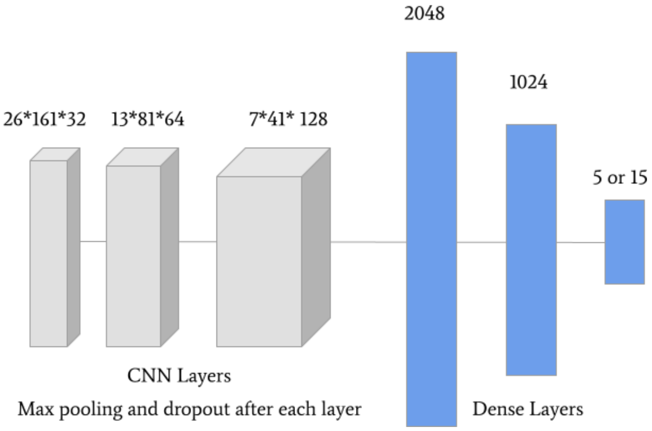


Figure 7. CNN architecture.

After feature extraction is performed by the convolutional layers, the results are fed into three dense layers for classification. Fewer neurons were used, as compared to the previous study, to prevent overfitting due to the smaller plot sizes. In order to find the most-optimal hyperparameters, a grid search was conducted on the values 0.01,0.001, and 0.0001 for the learning rate as well as 128, 256, and 512 for the batch sizes. The grid search ran all nine possibilities, and the most optimum learning rate and batch size were selected (Table 1). This SHM framework for the post-hazard management of nuclear safety systems was able to demonstrate its effectiveness in detecting degraded locations as well as the degradation severity.

Table 1. ANN training parameters.

Parameter	Value
Activation Functions	ReLu and SoftMax
Epochs	200
Learning Rate	0.001
Batch Size	256
Dropout	0.3
Optimization	ADAM
Validation Split	30%

On a traditional desktop machine (16Gb of RAM) with graphics-processing unit (GPU) capabilities, each epoch took about 30 minutes. Running 200 epochs on such a machine would be impractical and inefficient. The relative effectiveness of using a central processing unit, GPU, and tensor-processing unit (TPU) was explored to find a more-efficient solution. By implementing the code on Google Colaboratory (25 Gb of RAM) with a TPU, each epoch only took a few seconds. This opened up the possibilities of testing different CNN architectures and hyperparameters, as described above.

After selecting hyperparameters, the model predicted, from the data of nine sensors, degraded locations and degradation levels. From previous studies [15], it was known that degradation happens mostly on nozzles (anchors), elbows, and T-joints of a piping system, so sensors can be reduced to just the four on the elbows. The model predicted both locations and degradation levels, as well locations using only reduced sensor data.

5. RESULTS

An accuracy of 99% was achieved to detect degraded locations. The proposed framework was able to detect degraded locations along with their severity level with a 96% prediction accuracy. For the SHM of real structures and systems, it is not possible to put sensors at all locations. To decrease the economic and computational cost of dealing with a large number of sensors, as with the nine sensors detailed in the previous section, this research also explored the effects of a sensor-placement strategy. Only four sensors at the elbows, shown in Figure 8 were selected to acquire sensor response as time-series signals.

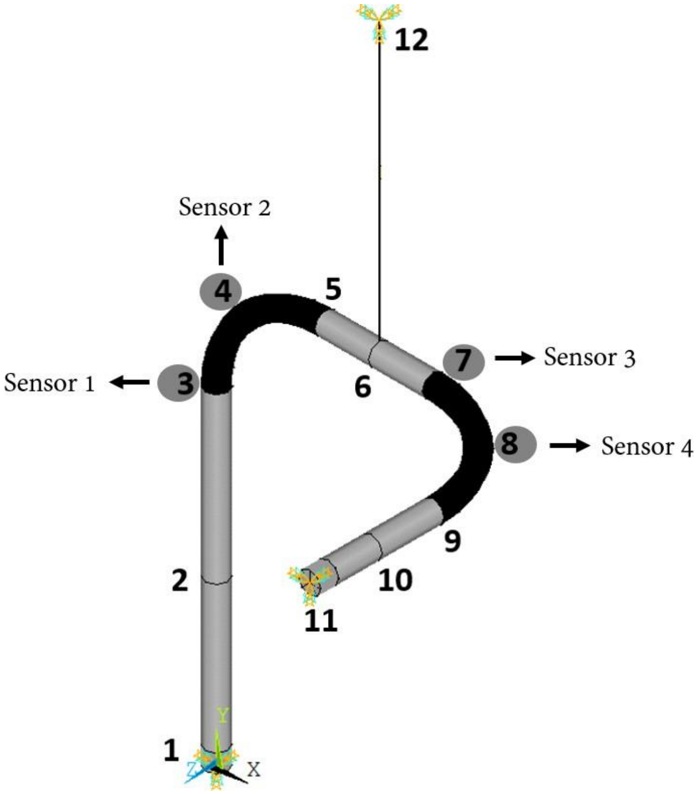


Figure 8. Sensor optimization.

The proposed SHM framework was able to predict degraded locations with 98% accuracy. A 97% accuracy was achieved when degradation severities were also added as the predicted output. It is observed that the proposed methodology is very effective for the health monitoring of the simple piping equipment system being considered. Even with reduced sensors, the CNN model was able to learn effectively, thereby increasing confidence in the selected sensor-placement strategy. The results from this application case study are tabulated in Table 2. When the dropout rate was increased for the 9 sensor framework, the model had slightly better results but it was not significant.

Table 2. Accuracy of proposed SHM framework.

Number of Sensors	Predict Locations	Predict Locations and Severity
9	99%	96%
4	98%	97%

It is observed that using only four sensors with this sensor-placement strategy yields better prediction accuracies. The phenomenon of overfitting can reduce predictive capability of any ANN model. Hence, in this case study, the amount of data acquired from nine sensors overfits the 2D CNN model when compared to the data acquired from only four sensors.

6. FUTURE WORK

Due to the larger amount of data when using nine sensors, it was expected that the nine-sensor model would have significantly better results than the four-sensor model. This was not seen in the results due to overfitting. To overcome this challenge, k-fold cross-validation technique can be applied in the future. In this method, the data would be split into k-many groups, and each group would be individually used as the test set, with the rest as a training set.

While this model was built from a simple piping system, it provides the foundation to be applied to realistic nuclear piping-equipment systems more-complex systems—such as the Experimental Breeder Reactor-II multibranched piping system—and to predicting degradation at multiple locations within a system.

7. ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to Dr. Piyush Sabharwall for guiding me throughout this project. His patient guidance and supportive encouragement during this internship has helped me develop as a researcher. The completion of this project would not be possible without the valuable support of Harleen Kaur Sandhu and Dr. Saran Srikanth Bodda. Their knowledge on the project and enthusiasm to help has changed my internship experience for the better. I am also extremely grateful to Dr. Abhinav Gupta for his unwavering support and his willingness to give his time. I would also like to thank the following people: Majdi I. Radaideh, Cody Walker, Jacob Farber, Yifeng Che, and Curtis Smith for hosting the MIT-INL AI/ML Summer 2021 Symposium. Lastly, I am deeply indebted to the Idaho National Laboratory and ORISE for allowing me to participate in this internship. This experience has meant a lot to me, and I appreciate this amazing opportunity given to me.

8. REFERENCES

[1] M. Grieves, "Digital Twin: Manufacturing Excellence through Virtual Factory Replication," (2014)URL http://www.apriso.com/library/Whitepaper_Dr_Grieves_DigitalTwin_ManufacturingExcellence.php.

[2] L. Lin, P. Athe, P. Rouxelin, M. Avramova, A. Gupta, R. Youngblood,

[3] J. Lane, and N. Dinh, "Development and assessment of a nearly autonomous management and control system for advanced reactors," Annals of Nuclear Energy, 150, 107861 (2021); 10.1016/j.anucene.2020.107861. https://linkinghub.elsevier. com/retrieve/pii/S0306454920305594.

[4] NAMAC, "Development of a Nearly Autonomous Management and Control (NAMAC) System for Advanced Reactors," (2018) https://arpa-e.energy.gov/?q=slick-sheet-project/management-and-control-system-advanced-reactors.

[5] B. R. Upadhyaya, K. Zhao, S. R. Perillo, X. Xu, and M. G. Na, "Autonomous Control of Space Reactor Systems,"; 10.2172/920996., URL https://www.osti.gov/ biblio/920996.

[6] R. T. Wood, "Autonomous Control for Generation IV Nuclear Plants," 6.

[7] H. Basher, "Autonomous Control of Nuclear Power Plants," ORNL/TM- 2003/252, 885601 (2003); 10.2172/885601., URL http://www.osti.gov/servlets/ purl/885601-TNIDBO/.

[8] Varuttamaseni, S. Yoo, and A. Borrelli, "Adaptive Control and Monitoring Platform for Autonomous Operation of Advanced Nuclear Reactors," 1.

[9] E. A. Patterson, R. J. Taylor, and M. Bankhead, "A framework for an integrated nuclear digital environment," Progress in Nuclear Energy, 87, 97 (2016); 10.1016/j.pnucene.2015.11.009., URL https://linkinghub.elsevier.com/retrieve/ pii/S0149197015301104.

[10] E. A. Patterson, S. Purdie, R. J. Taylor, and C. Waldon, "An integrated digital framework for the design, build and operation of fusion power plants," Royal Society Open Science, 6, 10,

181847 (2019); 10.1098/rsos.181847., URL https://royalsocietypublishing.org/doi/10.1098/rsos.181847.

[11] Gupta, H. K. Sandhu, S. S. Bodda, L. Lin, P. Athe, N. Dinh, J. Lane, and R. Youngblood, "Development of a Nearly Autonomous Management and Control System (NAMAC) for Advanced and Micro Reactors," SAIMIN: Symposium on Artificial Intelligence, Machine Learning and other Innovative Technologies in Nuclear Industry (2020).

[12] H. K. Sandhu, S. S. Bodda, and A. Gupta, "Structural Health Monitoring of Piping-Equipment Systems in Nuclear Power Plants using Artificial Neural Networks," 10th International Conference on Structural Health Monitoring of Intelligent Infrastructure (2020).

[13] P. C. Wu, "Erosion/Corrosion-Induced Pipe Wall Thinning in U.S. Nuclear Power Plants," NUREG–1344, 6152848 (1989); 10.2172/6152848., URL http://www.osti.gov/servlets/purl/6152848/.

[14] B. S. Ju and A. Gupta, "Seismic fragility of threaded Tee-joint connections in piping systems," International Journal of Pressure Vessels and Piping, 132-133, 106 (2015); 10.1016/j.ijpvp.2015.06.001., URL https://linkinghub.elsevier.com/retrieve/pii/S0308016115000708.

[15] Y. Ryu, A. Gupta, W. Jung, and B. Ju, "A Reconciliation of Experimental and Analytical Results for Piping Systems," International Journal of Steel Structures, 14 (2016).

[16] Gupta, Y. Ryu, and R. K. Saigal, "Performance-Based Reliability of ASME Piping Design Equations," Journal of Pressure Vessel Technology, 10 (2017).

[17] Y. Ryu, "Fragility of Piping Systems and Reliability of Piping Components," PhD Thesis, North Carolina State Univeristy (2013).

[18] M. Nifong, "Uncertainty of Threaded Piping Subjected to Monotonic Loading," PhD Thesis, North Carolina State Univeristy (2014).

[19] B. S. Ju, "Seismic Fragility of Piping System," PhD Thesis, North Carolina State Univeristy (2011).

[20] DATAmadness, "Time signal classification using CNN," https://github.com/datamadness/Time-signal-classification-using-Convolutional-Neural-Network (2019).

[21] P. Bezler, M. Hartzman, and M. Reich, "Piping Benchmark Problems," NUREG– 1677 (1980).

[22] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," Nature Methods, 17, 261 (2020); 10.1038/s41592-019-0686-2.

[23] J. B. Allen, "Short Time Spectral Analysis, Synthesis, and Modification by Discrete Fourier Transform," ASSP-25 (3), 235–238 (1977); 10.1109/TASSP.1977.1162950.

[24] G. Van Rossum, The Python Library Reference, release 3.8.2, Python Software Foundation (2020).

[25] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray,

C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems,” (2015)URL http://tensorflow.org/, software available from tensorflow.org.

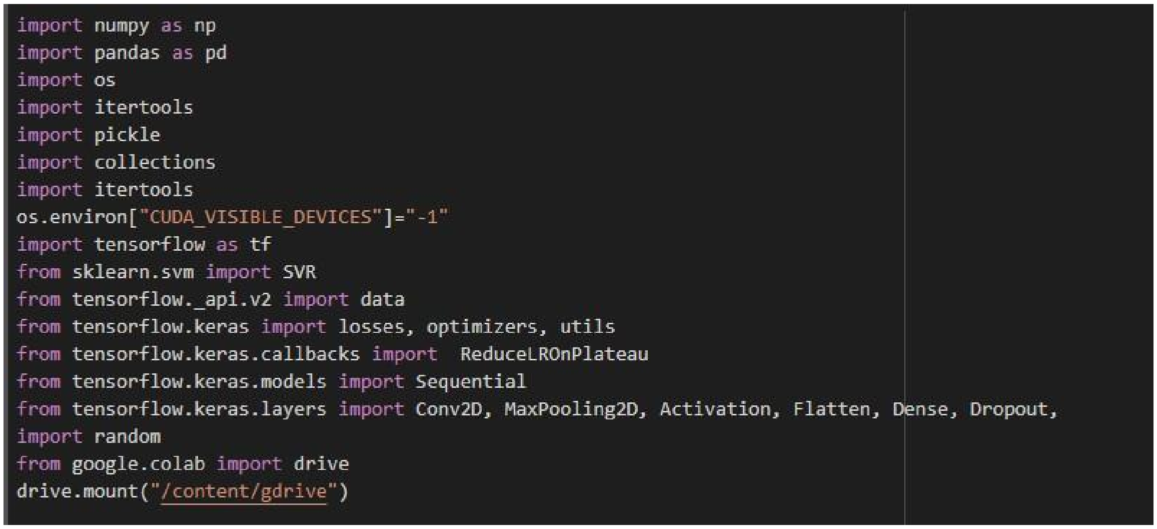
[26] F. Chollet, “Keras,” https://github.com/fchollet/keras (2015).

Appendix A

Creating the CNN

A-1. Initializing the Environment

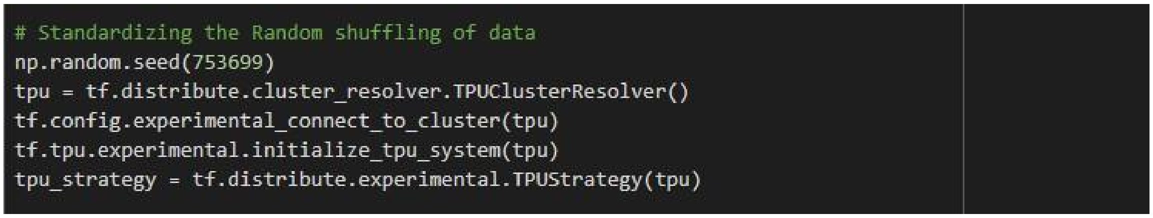
The first step in our code was to initialize what we needed to run in order to setup the data, neural network, and other data structures. Popular data analytic programs like Numpy, Pandas, and SkLearn were imported as well as Tensorflow and Keras functions. Then Google Drive was mounted in order to access the data (Figure 9).



```
import numpy as np
import pandas as pd
import os
import itertools
import pickle
import collections
import itertools
os.environ["CUDA_VISIBLE_DEVICES"]="-1"
import tensorflow as tf
from sklearn.svm import SVR
from tensorflow._api.v2 import data
from tensorflow.keras import losses, optimizers, utils
from tensorflow.keras.callbacks import  ReduceLROnPlateau
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Activation, Flatten, Dense, Dropout,
import random
from google.colab import drive
drive.mount("/content/gdrive")
```

Figure 9. Importing analytic programs and mounting Google Drive.

Because data for testing and training need to eventually be shuffled, we introduced a random seed so the random shuffle is standardized. Then we connected the file to the built in Google Colaboratory TPU for a faster runtime (Figure 10).

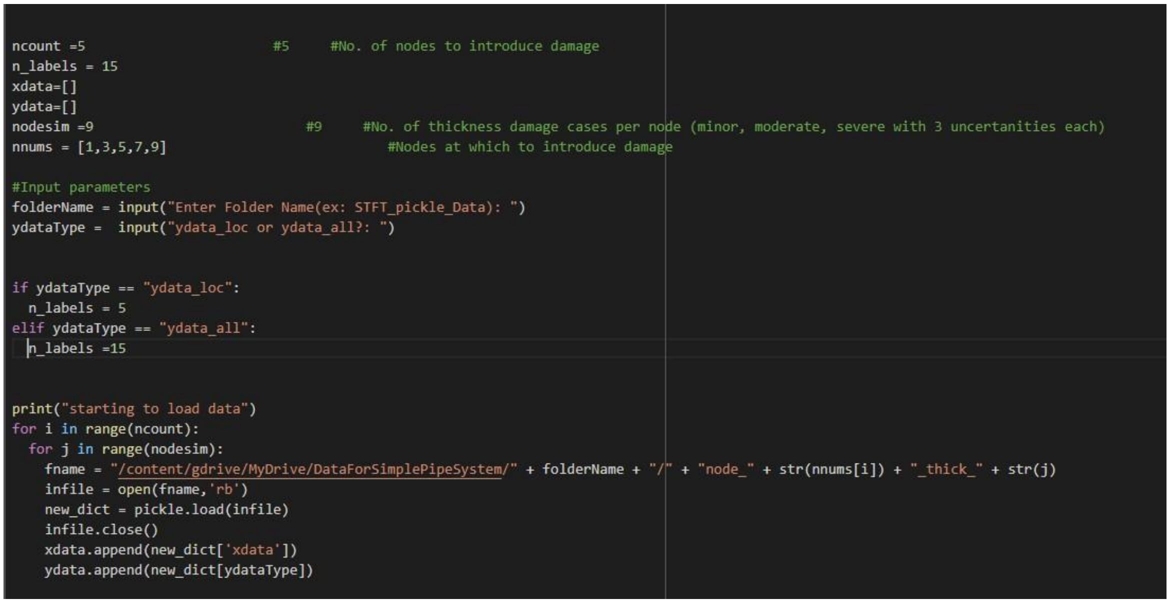


```
# Standardizing the Random shuffling of data
np.random.seed(753699)
tpu = tf.distribute.cluster_resolver.TPUClusterResolver()
tf.config.experimental_connect_to_cluster(tpu)
tf.tpu.experimental.initialize_tpu_system(tpu)
tpu_strategy = tf.distribute.experimental.TPUStrategy(tpu)
```

Figure 10. Standardizing random shuffling of the data.

A-2. Data Preprocessing

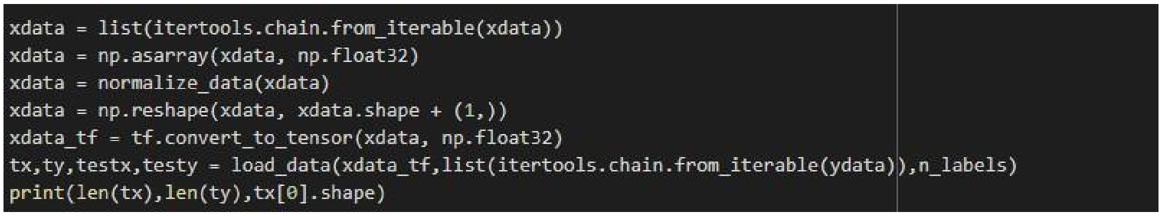
Because we need to introduce damage to five nodes, ncount, the variable for node count, is set to 5. Then, depending on whether we are testing for all data for just the location the number of labels, num_labels is set to 15 (for location and degradation of 5 locations * 3 degradation levels) or five (for locations only). Then the Pickle data from the folder are iterated through twice for loops and appended into xdata and ydata (Figure 11).



```
ncount =5                       #5     #No. of nodes to introduce damage
n_labels = 15
xdata=[]
ydata=[]
nodesim =9                          #9     #No. of thickness damage cases per node (minor, moderate, severe with 3 uncertanities each)
nnums = [1,3,5,7,9]                          #Nodes at which to introduce damage

#Input parameters
folderName = input("Enter Folder Name(ex: STFT_pickle_Data): ")
ydataType =  input("ydata_loc or ydata_all?: ")


if ydataType == "ydata_loc":
  n_labels = 5
elif ydataType == "ydata_all":
  n_labels =15


print("starting to load data")
for i in range(ncount):
  for j in range(nodesim):
    fname = "/content/gdrive/MyDrive/DataForSimplePipeSystem/" + folderName + "/" + "node_" + str(nnums[i]) + "_thick_" + str(j)
    infile = open(fname,'rb')
    new_dict = pickle.load(infile)
    infile.close()
    xdata.append(new_dict['xdata'])
    ydata.append(new_dict[ydataType])
```

Figure 11. Data iteration.

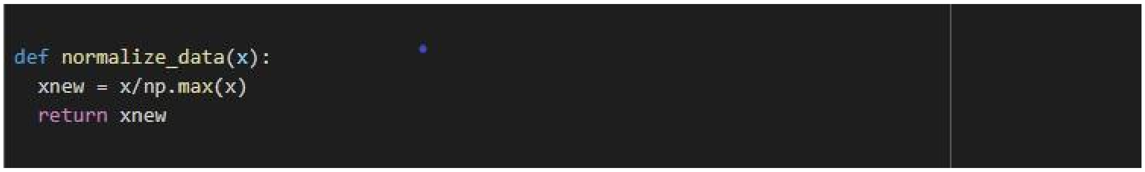
The data is flattened using itertools.chain.from_itertools() so it can be easily reshaped. Then items in that list were converted to a numpy array with float values. The data were then normalized and then reshaped to fit the CNN model and converted to a tensor so it can be inputted into a neural network later (Figure 12). The data is then split into testing and training data.



```
xdata = list(itertools.chain.from_iterable(xdata))
xdata = np.asarray(xdata, np.float32)
xdata = normalize_data(xdata)
xdata = np.reshape(xdata, xdata.shape + (1,))
xdata_tf = tf.convert_to_tensor(xdata, np.float32)
tx,ty,testx,testy = load_data(xdata_tf,list(itertools.chain.from_iterable(ydata)),n_labels)
print(len(tx),len(ty),tx[0].shape)
```

Figure 12. Data flattening.

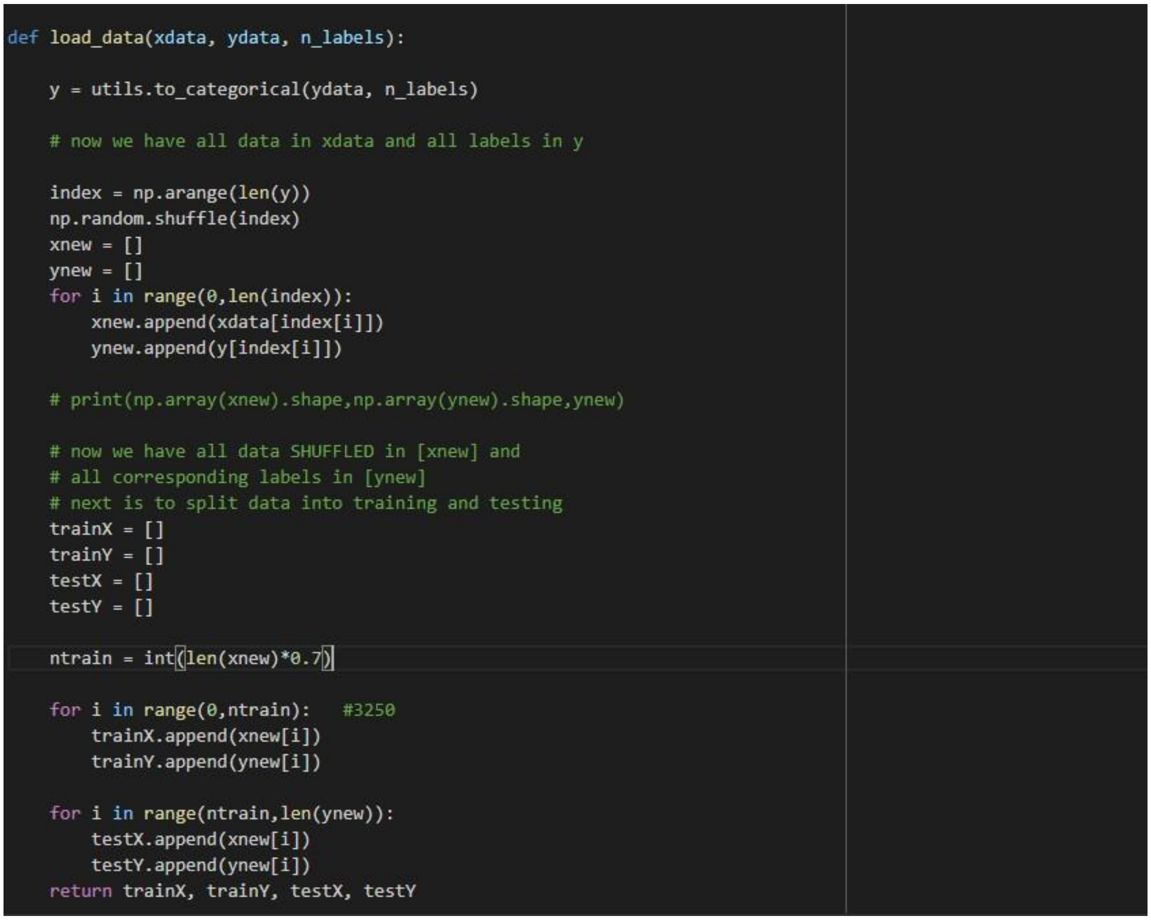
In order to normalize the data we divided all values with respect to the maximum value. This creates values between 0 and 1 for the inputs in the neural network (Figure 13). Because reLU ignores values under 0 and keeps values above 0, if a number is negative, data may be lost, but if a value is too large, it may be too-heavily weighted.



```
def normalize_data(x):
  xnew = x/np.max(x)
  return xnew
```

Figure 13. Data normalization.

In this function, xdata and ydata are shuffled and then split into training and testing data. Seventy percent of the data is set for training data, and 30% for testing. The testing data for x and y are then put into testX and testY respectively. Likewise, the training data for x and y were put into trainX and trainY (Figure 14).

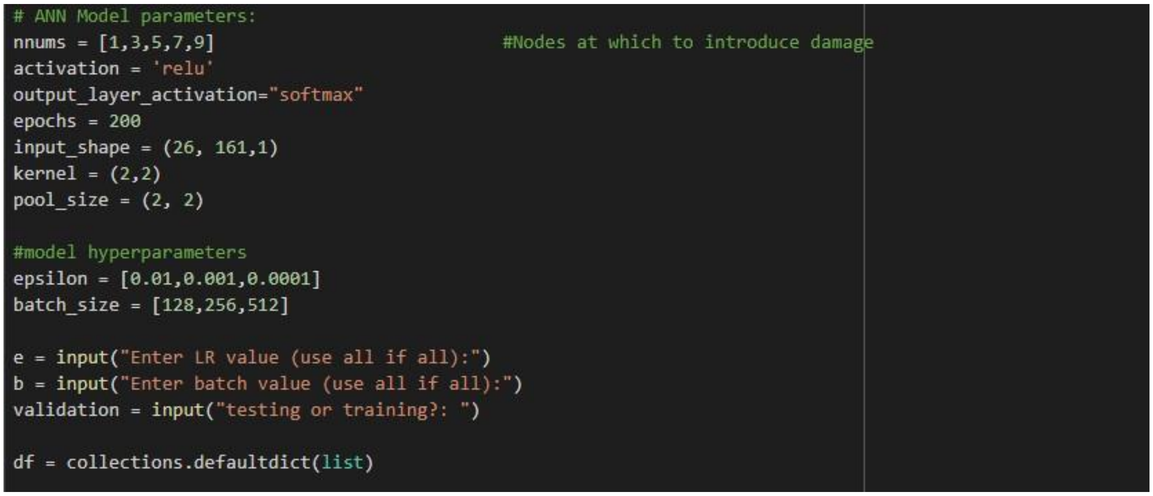


```
def load_data(xdata, ydata, n_labels):

    y = utils.to_categorical(ydata, n_labels)

    # now we have all data in xdata and all labels in y

    index = np.arange(len(y))
    np.random.shuffle(index)
    xnew = []
    ynew = []
    for i in range(0,len(index)):
        xnew.append(xdata[index[i]])
        ynew.append(y[index[i]])

    # print(np.array(xnew).shape,np.array(ynew).shape,ynew)

    # now we have all data SHUFFLED in [xnew] and
    # all corresponding labels in [ynew]
    # next is to split data into training and testing
    trainX = []
    trainY = []
    testX = []
    testY = []

    ntrain = int(len(xnew)*0.7)

    for i in range(0,ntrain):   #3250
        trainX.append(xnew[i])
        trainY.append(ynew[i])

    for i in range(ntrain,len(ynew)):
        testX.append(xnew[i])
        testY.append(ynew[i])
    return trainX, trainY, testX, testY
```

Figure 14. Separating data between training and testing groups.

A-3. Building the CNN

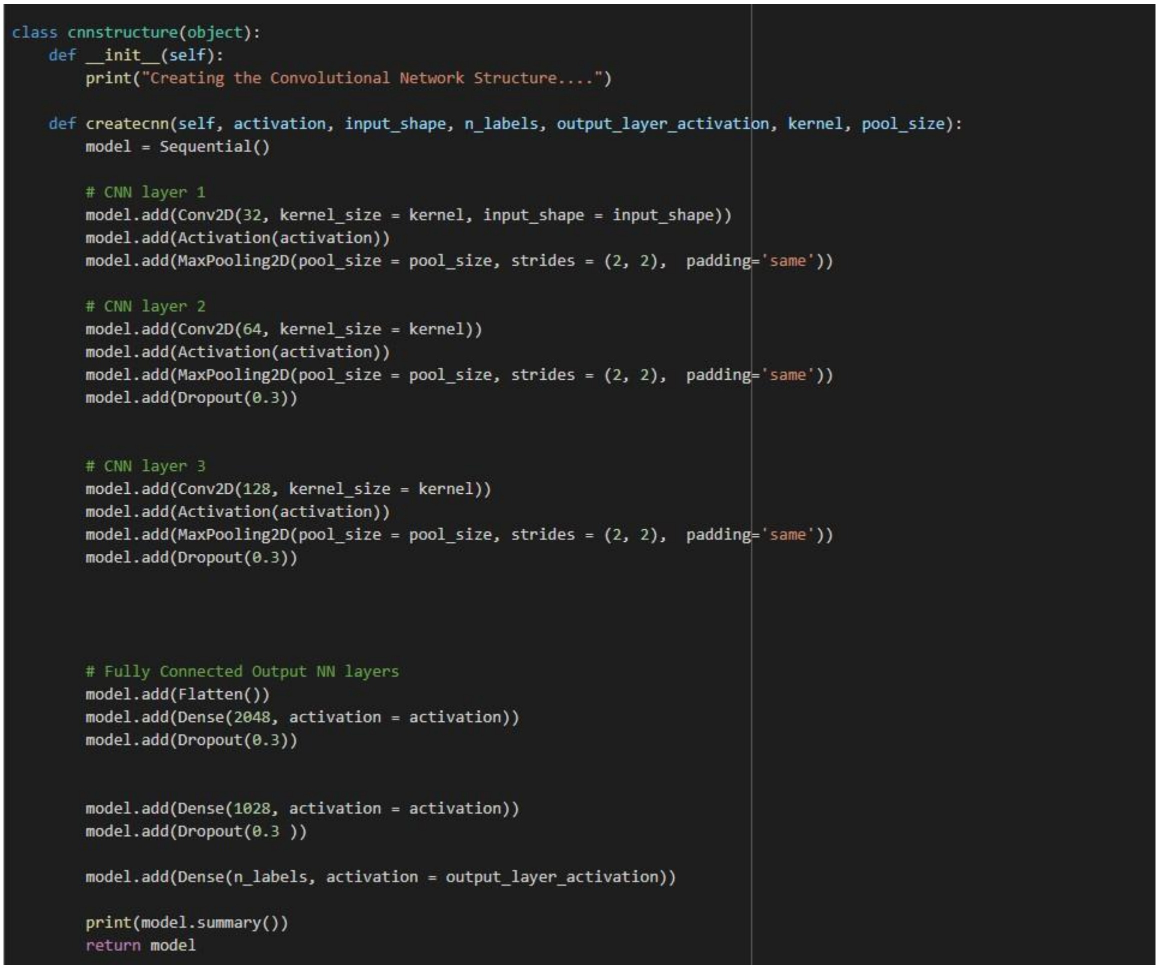
The parameter that needed to be initialized were set such as the activation functions, pool size, kernel size, and number of epochs. There is an option for user input for hyperparameters such as learning rate and batch size for use during testing (Figure 15).



```
# ANN Model parameters:
nnums = [1,3,5,7,9]                        #Nodes at which to introduce damage
activation = 'relu'
output_layer_activation="softmax"
epochs = 200
input_shape = (26, 161,1)
kernel = (2,2)
pool_size = (2, 2)

#model hyperparameters
epsilon = [0.01,0.001,0.0001]
batch_size = [128,256,512]

e = input("Enter LR value (use all if all):")
b = input("Enter batch value (use all if all):")
validation = input("testing or training?: ")

df = collections.defaultdict(list)
```

Figure 15. Setting parameters and hyperparameters.

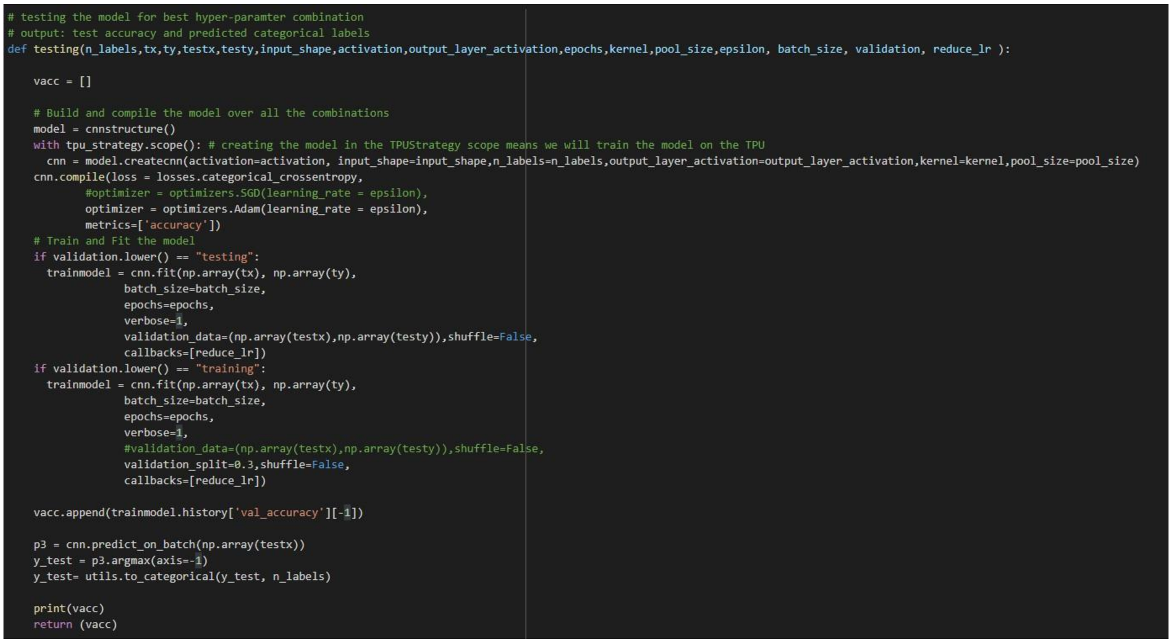
Our CNN structure uses 2D convolution layers as well as max pooling, dropout, and dense layers. The CNN has three convoluted layers, each with each subsequent layer-doubling in the number of neurons. All three layers use reLU as an activation function. There is a max-pooling layer after each convoluted layer to emphasize the features extracted in the convoluted layer. A dropout is used to prevent overfitting in the data. Then the output of the convoluted layers is fed into dense layers so that the dense layers can find the classifications after the convoluted layers extract the features (Figure 16).



```
class cnnstructure(object):
    def __init__(self):
        print("Creating the Convolutional Network Structure....")

    def createcnn(self, activation, input_shape, n_labels, output_layer_activation, kernel, pool_size):
        model = Sequential()

        # CNN layer 1
        model.add(Conv2D(32, kernel_size = kernel, input_shape = input_shape))
        model.add(Activation(activation))
        model.add(MaxPooling2D(pool_size = pool_size, strides = (2, 2),  padding='same'))

        # CNN layer 2
        model.add(Conv2D(64, kernel_size = kernel))
        model.add(Activation(activation))
        model.add(MaxPooling2D(pool_size = pool_size, strides = (2, 2),  padding='same'))
        model.add(Dropout(0.3))


        # CNN layer 3
        model.add(Conv2D(128, kernel_size = kernel))
        model.add(Activation(activation))
        model.add(MaxPooling2D(pool_size = pool_size, strides = (2, 2),  padding='same'))
        model.add(Dropout(0.3))




        # Fully Connected Output NN layers
        model.add(Flatten())
        model.add(Dense(2048, activation = activation))
        model.add(Dropout(0.3))


        model.add(Dense(1028, activation = activation))
        model.add(Dropout(0.3 ))

        model.add(Dense(n_labels, activation = output_layer_activation))

        print(model.summary())
        return model
```

Figure 16. Creating the CNN.

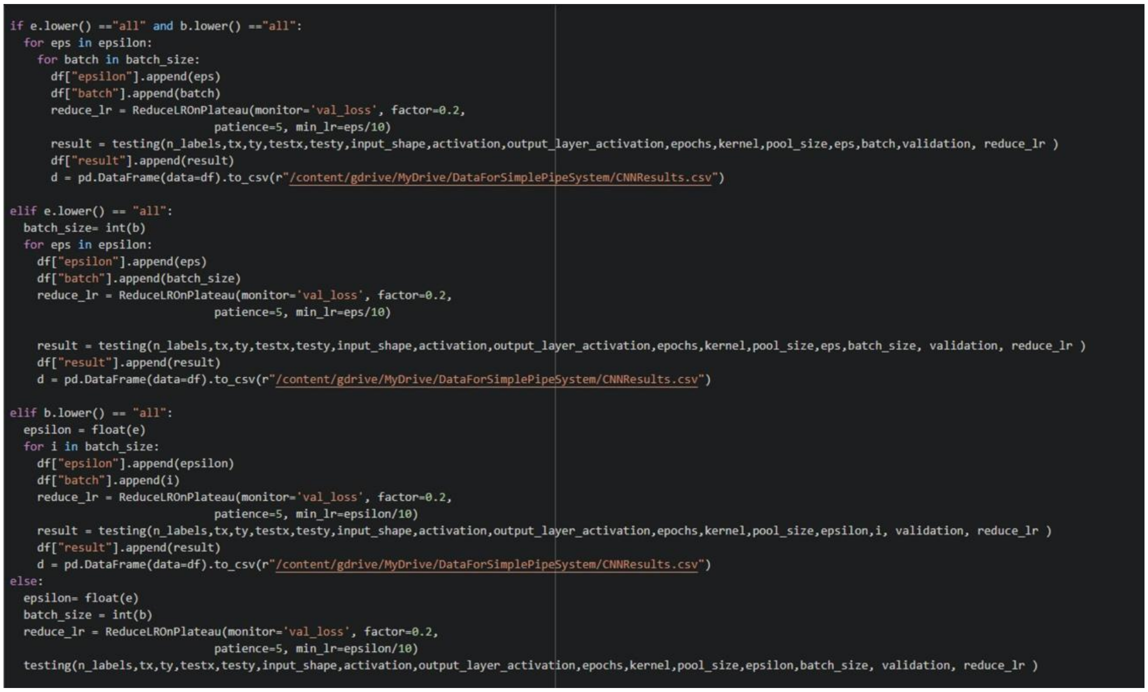
The testing function is what calls the cnnstructure function. In here, we set the optimizer to Adam and the learning rate based on what we selected in the first code block of this section. Then, depending on testing or training, we run the model on a validation split or testing data (Figure 17). The CNN is called under the TPU strategy scope so the code will be trained under the TPU.



```python
# testing the model for best hyper-paramter combination
# output: test accuracy and predicted categorical labels
def testing(n_labels,tx,ty,testx,testy,input_shape,activation,output_layer_activation,epochs,kernel,pool_size,epsilon, batch_size, validation, reduce_lr ):

    vacc = []

    # Build and compile the model over all the combinations
    model = cnnstructure()
    with tpu_strategy.scope(): # creating the model in the TPUStrategy scope means we will train the model on the TPU
      cnn = model.createcnn(activation=activation, input_shape=input_shape,n_labels=n_labels,output_layer_activation=output_layer_activation,kernel=kernel,pool_size=pool_size)
    cnn.compile(loss = losses.categorical_crossentropy,
            #optimizer = optimizers.SGD(learning_rate = epsilon),
            optimizer = optimizers.Adam(learning_rate = epsilon),
            metrics=['accuracy'])
    # Train and Fit the model
    if validation.lower() == "testing":
      trainmodel = cnn.fit(np.array(tx), np.array(ty),
                  batch_size=batch_size,
                  epochs=epochs,
                  verbose=1,
                  validation_data=(np.array(testx),np.array(testy)),shuffle=False,
                  callbacks=[reduce_lr])
    if validation.lower() == "training":
      trainmodel = cnn.fit(np.array(tx), np.array(ty),
                  batch_size=batch_size,
                  epochs=epochs,
                  verbose=1,
                  #validation_data=(np.array(testx),np.array(testy)),shuffle=False,
                  validation_split=0.3,shuffle=False,
                  callbacks=[reduce_lr])

    vacc.append(trainmodel.history['val_accuracy'][-1])

    p3 = cnn.predict_on_batch(np.array(testx))
    y_test = p3.argmax(axis=-1)
    y_test= utils.to_categorical(y_test, n_labels)

    print(vacc)
    return (vacc)
```

Figure 17. Running the model.

Depending on the batch size and learning rate we choose, we run all the CNN code above with the correct parameters.



```python
if e.lower() =="all" and b.lower() =="all":
  for eps in epsilon:
    for batch in batch_size:
      df["epsilon"].append(eps)
      df["batch"].append(batch)
      reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2,
                            patience=5, min_lr=eps/10)
      result = testing(n_labels,tx,ty,testx,testy,input_shape,activation,output_layer_activation,epochs,kernel,pool_size,eps,batch,validation, reduce_lr )
      df["result"].append(result)
      d = pd.DataFrame(data=df).to_csv(r"/content/gdrive/MyDrive/DataForSimplePipeSystem/CNNResults.csv")

elif e.lower() == "all":
  batch_size= int(b)
  for eps in epsilon:
    df["epsilon"].append(eps)
    df["batch"].append(batch_size)
    reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2,
                            patience=5, min_lr=eps/10)

    result = testing(n_labels,tx,ty,testx,testy,input_shape,activation,output_layer_activation,epochs,kernel,pool_size,eps,batch_size, validation, reduce_lr )
    df["result"].append(result)
    d = pd.DataFrame(data=df).to_csv(r"/content/gdrive/MyDrive/DataForSimplePipeSystem/CNNResults.csv")

elif b.lower() == "all":
  epsilon = float(e)
  for i in batch_size:
    df["epsilon"].append(epsilon)
    df["batch"].append(i)
    reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2,
                            patience=5, min_lr=epsilon/10)
    result = testing(n_labels,tx,ty,testx,testy,input_shape,activation,output_layer_activation,epochs,kernel,pool_size,epsilon,i, validation, reduce_lr )
    df["result"].append(result)
    d = pd.DataFrame(data=df).to_csv(r"/content/gdrive/MyDrive/DataForSimplePipeSystem/CNNResults.csv")
else:
  epsilon= float(e)
  batch_size = int(b)
  reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2,
                            patience=5, min_lr=epsilon/10)
  testing(n_labels,tx,ty,testx,testy,input_shape,activation,output_layer_activation,epochs,kernel,pool_size,epsilon,batch_size, validation, reduce_lr )
```

Figure 18. Final data run.