Application of a U-Net Convolutional Neural Network to Ultrasonic Wavefield Measurements for Defect Characterization

Joshua D. Eckels[1], Isabel F. Fernandez[1], Kelly Ho[1],
Nikolaos Dervilis[2], Erica M. Jacobson[1], and Adam J. Wachtor[1]

[1]NSEC-EI, Los Alamos National Laboratory,
Los Alamos, NM 87545

[2]Department of Mechanical Engineering, The University of Sheffield,
Sheffield, UK S10 2TN

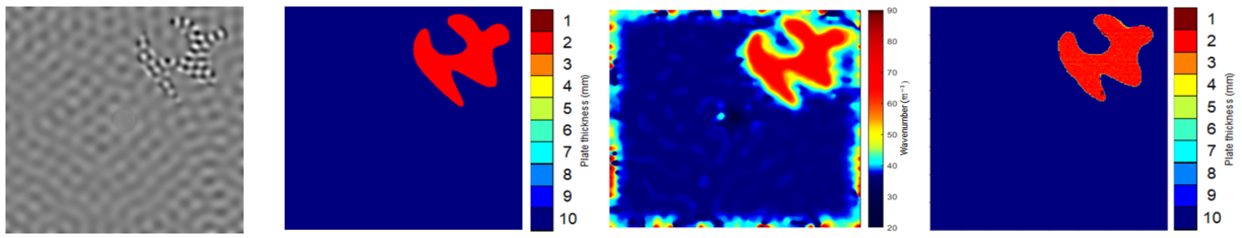
August 25, 2020

1 Project information

The goal of this project is to utilize a U-net style convolutional neural network (CNN) to identify and characterize defects in a plate-like structure. Acoustic wavenumber spectroscopy (AWS) is the current system that takes a 2D, steady-state, out-of-plane surface velocity response of a plate-like structure to a single-tone ultrasonic excitation, (via attached transducer), and identifies defects in the plate based on local changes in wavenumber. Local changes in wavenumber correspond to local changes in plate thickness via Lamb-wave dispersion relationships. Local changes in thickness of the plate correspond to damage and defects, such as corrosion, internal cracking, or delamination. In essence, the output of the CNN is comparable to the output of AWS in that it visually locates defects in a plate and provides some quantification of its relative severity. In contrast to AWS, the CNN directly predicts the local thickness of the plate rather than the wavenumber. In addition, the CNN improves upon the current AWS algorithm by providing considerable speedup in processing time (AWS: 8 seconds ; CNN: 0.1 seconds), as well as consistency in predicting defects along the edges of the scan region.

This project, at its very least, provides a proof-of-concept that a CNN can be trained to recognize patterns and features in an ultrasonic wavefield, and that it can provide comparable, if not better results to the current AWS algorithm. It is our belief that with further improvement, the CNN can consistently produce better results where the AWS algorithm is currently limited (and faster), as well as prove to be very robust with generalization to a greater variety of defects in a greater variety of structures and components. We believe the CNN should be developed further and take a more prominent role in the field of using steady-state ultrasonic excitation for non-destructive evaluation.

The purpose of this document is to thoroughly record and document all setup and procedures of the project at the time of writing, as well as details not included in the published paper, in order that the reader may replicate the work and continue it further. Figure 1 below shows one of the prominent outputs of the CNN in its current state in comparison to AWS. This document will walk through all procedures needed to recreate this result.



(a) Wavefield image **(b)** Ground truth **(c)** AWS prediction **(d)** CNN prediction

Figure 1: AWS v. CNN results for an irregular shape

Contents

2 Project overview

The project overview map shown in figure 2 provides an ordered high-level view of the many tasks performed in this project.

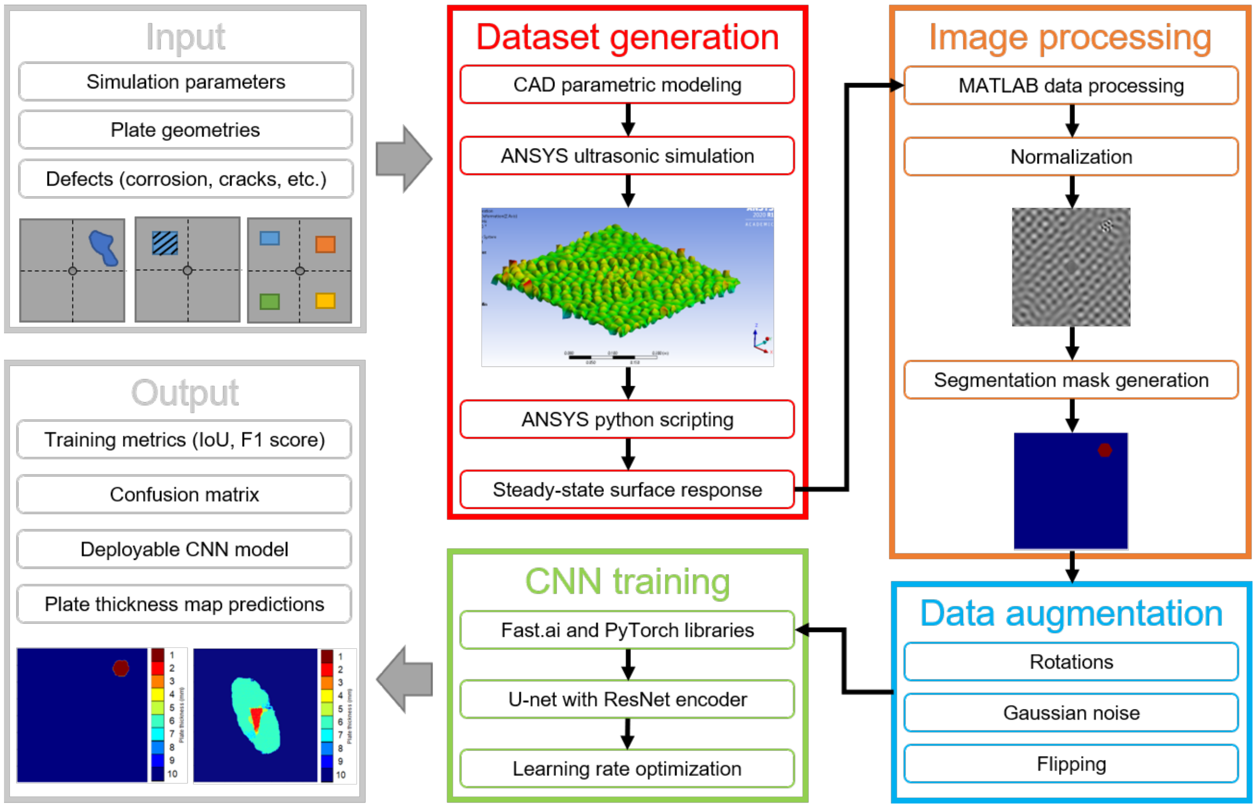

Input
Simulation parameters
Plate geometries
Defects (corrosion, cracks, etc.)
Dataset generation
CAD parametric modeling
ANSYS ultrasonic simulation
ANSYS python scripting
Steady-state surface response
Image processing
MATLAB data processing
Normalization
Segmentation mask generation
Output
Training metrics (IoU, F1 score)
Confusion matrix
Deployable CNN model
Plate thickness map predictions
CNN training
Fast.ai and PyTorch libraries
U-net with ResNet encoder
Learning rate optimization
Data augmentation
Rotations
Gaussian noise
Flipping

Figure 2: Project overview map

The next section will walk through each of these steps in greater detail. In general,

1. **Input:** A simulation scheme was brainstormed by hand that tried to incorporate as many scenarios in the dataset as possible. The final trained CNN model is only as good as the data it is given, so this is perhaps the most important step for improving the CNN model's accuracy and robustness. Among the parameters that must be chosen are excitation frequency, plate dimensions, size and shape of defects, quantity and quality of defects, location and thickness of defects, transducer location, and mesh size and other simulation parameters.

2. **Dataset generation:** Once a simulation scheme was developed on paper, the parameters must be input into CAD modeling to generate hundreds of CAD *.step* geometry files. SpaceClaim was used within ANSYS workbench to generate these models in a *for* loop by iteratively adjusting different dimensions. Most other CAD software packages can perform similar parametric modeling. Once the models were generated, a Python script was used to run hundreds of ANSYS harmonic response simulations on the geometry *.step* files. The real and imaginary parts of the plate's surface deformation response (representing the steady-state simple sinusoidal oscillations at every mesh node point) were exported to *.txt* files by ANSYS ADPL commands.

3. **Image processing:** The real and imaginary parts of the steady-state deformation response of the plate were then read directly into a MATLAB script for further processing. MATLAB was used to filter out surface mesh points, interpolate to an evenly-spaced grid, normalize the data, and generate a grayscale wavefield image for all of the real, imaginary, and magnitude steady-state response of the plate. MATLAB also was used to automatically generate segmentation masks of the plate indicating the location and thickness of defects on a pixel by pixel basis.

4. **Data augmentation:** The *numpy* and *scikit-learn* Python libraries were used to rotate and add Gaussian noise to all of the wavefield images to increase the dataset size.

5. **CNN training:** The *Fast.ai* and *PyTorch* machine learning Python libraries were used to build and train a U-net CNN architecture for an image segmentation task. The popular *ResNet* architecture was pre-trained on the ImageNet database and loaded as an encoder into our U-net model. The *Fast.ai* recommended protocols for training a CNN model were followed where applicable. *Fast.ai* follows the *fit one cycle* mantra, where the CNN model with pretrained weights and biases is frozen and the last few fully-connected layers are trained on the new dataset (our wavefield dataset) using an optimal learning rate. The CNN model is then mostly unfrozen and trained again on our dataset to refine the internal layers.

6. **Output:** The trained CNN model reports its final intersection-over-union (IoU) value and F1 score (dice coefficient). These training accuracy metrics can be used to benchmark performance against other published CNN models. The trained CNN model can be exported to a pickle (*.pkl*) file to be deployed on new systems for running inference on new images. The output of running inference on a new image is a color-coded segmentation mask that indicates the predicted local plate thickness on a pixel by pixel level.

3 Detailed project workflow

This section will seek to provide a more comprehensive set of procedures that can be followed to replicate this project from scratch. It will also point out common pitfalls we experienced along the way, as well as existing problems that still need to be addressed.

3.1 Software and installation details

All software used in this project is detailed in table 1 below, as well as installation and documentation links. All dataset generation and image processing were performed on a Windows 10 system; data augmentation and CNN training were performed on a Debian-based Linux virtual machine.

Table 1: Software and hardware details and installation links

Name	Version	Link
Python	3.7.6	`https://docs.anaconda.com/anaconda/install/`
Fast.ai	1.0.61	`https://github.com/fastai/fastai1`
PyTorch	1.4.0	(installed with Fast.ai)
numpy	1.18.1	(installed with Fast.ai)
matplotlib	3.2.1	(installed with Fast.ai)
PIL	7.1.2	(installed with Fast.ai)
Jupyter notebook	-	(installed with Fast.ai)
skimage	0.17.2	`https://scikit-image.org/docs/dev/install.html`
NVIDIA driver	418.87	NVIDIA Tesla P100-PCIE-16GB GPU
CUDA	10.1	`https://developer.nvidia.com/cuda-zone`
MATLAB	R2019a	Statistics, Parallel computing toolbox, Engine API for Python
ANSYS	19.1	`https://www.ansys.com/academic`
Dropbox	10.3.0	`https://www.dropbox.com/developers/documentation`

ANSYS and MATLAB licenses are required (proprietary software). The Statistics toolbox for MATLAB is required for segmentation mask generation. The Parallel Computing toolbox is recommended for processing speed-up, but not required. The MATLAB Engine for Python was used to further automate the workflow between ANSYS simulations and MATLAB post-processing, but this is not required. ANSYS scripting uses the IronPython 2.7 version of the generic Python language. The Anaconda distribution of Python may be used for all deep learning software tools, (not required; you can use a generic Python installation). The Dropbox API can be used with a free or paid subscription. The NVIDIA GPU and CUDA toolkit were used on a cloud computing virtual machine through Google Cloud Platform (GCP). GCP is a paid service with a free trial period; you may use this or any other suitable hardware acceleration platform for deep learning (Google Collab, Kaggle, personal computer, etc.). More details on setting up a GCP deep learning instance with Fast.ai are included in the *project_workflow.pptx* slides in the documentation folder. The Fast.ai Python library and all dependencies (PyTorch, numpy, PIL, etc.) may be installed with the conda or pip package managers (see the Fast.ai installation link). It is recommended that these are installed on your CUDA-enabled machine within a Python virtual environment (whether through Anaconda or another virtual environment tool, like virtualenv).

3.2 Directory structure

Most of the source code files used in the project are heavily dependent on the underlying directory structure. Not only will this directory structure keep your project organized, but it will allow source code files to access the data they need to run. Everything is coded relative to the project's root folder, which can be placed anywhere in the user's home filesystem. The following list outlines the project's directory structure and its usage.

/DeepWaves	*DeepWaves project root folder*
/ansys	*ANSYS project folder*
/DeepWaves_files	*ANSYS generated file storage for .wbpj*
/dp0/SYS/MECH	*Location of ADPL output data.txt files*
real.txt	*Real part of steady-state deformation response*
imaginary.txt	*Imaginary part of steady-state deformation response*
DeepWaves.wbpj	*ANSYS workbench project file; run Python script here*
/data	*MATLAB reads all data.txt files from this folder*
example_sim_real.txt	*Location where real.txt is moved and renamed*
example_sim_imaginary.txt	*Location where imaginary.txt is moved and renamed*
another_sim_real.txt	
another_sim_imaginary.txt	
/documentation	*All doc files like this one*
/geometry	*ANSYS reads all .step geometry files from here*
example_sim.step	
another_sim.step	
/images	*MATLAB outputs wavefield images here and*
example_sim_real.png	*CNN reads image dataset from here*
example_sim_imaginary.png	
example_sim_magnitude.png	
another_sim_real.png	
another_sim_imaginary.png	
another_sim_magnitude.png	
/labels	*MATLAB outputs segmentation masks here and*
example_sim_mask.png	*CNN reads segmentation masks from here*
another_sim_mask.png	
/logs	*Store all log files during script execution*
run_sims.log	*ANSYS simulation log file*
/mat	*Store all complex deformation matrices as .mat files here*
example_sim_vqz.mat	
another_sim_vqz.mat	
/models	*CNN exports model .pth and .pkl files here*
training_run.pth	*.pth files rely on access to dataset to reload CNN model*
training_run.pkl	*.pkl files are deployable trained CNN models*
/src	*All Python and MATLAB source code files here*
/CAD_scripts	*Directory containing ACT CAD modeling code snippets*
ACT_mech_script.py	*Runs harmonic response sim in ANSYS ACT console*
augment.py	*Creates new augmented images from dataset*
codes.txt	*Read by CNN and MATLAB to enumerate plate thickness classes*

display_dir.m	*MATLAB function to display images in a folder*
filter_defect.m	*Uses kmeans clustering to find plate defects in ANSYS node data*
noise.py	*Script to add Gaussian noise to images*
norm_batch.m	*Batch normalization over entire dataset*
plot_wavefield.m	*Generates wavefield images and segmentation masks*
process_images.m	*Calls plot_wavefield.m over entire dataset*
run_matlab.py	*Uses MATLAB Python engine to call .m files*
run_sims.py	*Automates ANSYS simulation process*
unet_train_JE.ipynb	*Jupyter notebooks to train CNN : JE method*
unet_train_0_004_KH.ipynb	*KH method*
unet_train_0_0009_KH.ipynb	*KH method*
/test	*CNN runs inference on all test images in here*
test_image_real.png	
another_test.png	

It is noted that not all features are tested across different systems. Different installations and versions may cause different problems in running the code. All code used in running ANSYS simulations especially might need to be tweaked depending on your system. It is also unsure how well ANSYS project files translate between systems (all details for setup are provided below though). Data exported from ANSYS is heavy in memory requirements. If a large number of simulations are meant to be run overnight, one might consider setting up a cloud storage account with Dropbox and utilizing the Dropbox API to automatically move simulation files off of the host computer to prevent filling up or crashing the hard drive, (run_matlab.py has some inspiration for this task if desired). The *directory_structure.txt* and *source_files.txt* files in the documentation folder provide more detail on this section.

3.3 Input

Table 2 shows all of the parameters accounted for in this study. Several parameters, like plate geometry and material, were kept constant throughout the study. A total of 8 “rounds” of datasets were generated by deciding how and which parameters changed in each round.

Table 2: Geometry and simulation parameters for dataset generation

Parameter	**Value**
Mesh size	2 mm
Plate size	400×400 mm
Plate Material	Aluminum alloy (ANSYS generic)
Plate thickness	$1 - 10$ mm
Transducer frequency	80 kHz
Transducer location	Variable
Number of defects	$1 - 4$
Plate thickness at defect	$1 - 9$ mm
Defect location	Variable
Defect shape	Variable
Defect size	Variable

The *simulation_tracker.xlsx* spreadsheet and the *simulation_plan.pptx* slides provide more detail on the 8 rounds of simulations performed in this study. Future studies should account for a greater variety of the dataset parameters listed above, including plate material, size, and transducer frequency. It is also noted here that there are several unaccounted parameters not listed here that could be studied in the future, including the shape, size, and number of transducers.

3.4 Dataset generation

Once a simulation scheme is developed by hand from the previous "Input" section, there are 3 main steps in generating the physical data from the devised simulation scheme: modeling the geometry in computer-aided design (CAD) software, setting up a finite element analysis (FEA) solver, and automating the FEA solver.

3.4.1 CAD parametric modeling

Most CAD software packages (Solidworks, Inventor, SpaceClaim, etc.) have a way for users to set certain geometry dimensions (length, depth, etc.) as variable parameters that are programmatically varied to generate multiple geometry models in an incremental fashion. This form of parametric modeling was performed in SpaceClaim in this study to quickly generate all of the geometry models devised in the simulation scheme (usually on the order of 100-200 unique geometries per round). If the reader is using a different CAD package, they might seek the appropriate resources to perform similar modeling in their respective CAD package.

In the SpaceClaim modeler provided within ANSYS workbench, ANSYS offers the *ANSYS Customization Toolkit* (ACT) console for interactive Python programming within the CAD software. The *src/CAD_scripts* directory contains several ACT Python code snippets that were run directly within the ACT console to generate all of the geometry *.step* files used in this study. The reader may use these code snippets as references in writing their own parametric modeling scripts within SpaceClaim to generate their unique geometry files for their simulation schemes. It is noted here that documentation online for the ACT Python API is very limited; the best way to quickly write working Python code within the ACT console is to use the journaling feature provided within ANSYS to record all interaction with the CAD graphical user interface (GUI) and generate Python code to replicate all recorded GUI interactions (mouse clicks, opening menus, selecting faces, etc.).

For the sake of simplicity and compliance with the ANSYS simulation described in the next sections, each geometry should be modeled as a plate-like structure on the XY plane with its center at the 3D origin of the CAD software design space, and the depth of the plate extruded outward in the positive z-direction. There should be a small cylindrical extrusion of 1 mm in the positive z-direction off of the plate's top face. This small cylinder is meant to model the physical ultrasonic transducer on the plate (it was given a diameter of 40 mm in this study). The simulations assume the top face of the transducer is located at the greatest z-coordinate value in the geometry (always true if you follow the setup described here). Figure 3 shows a screenshot of a typical plate geometry modeled in this fashion.

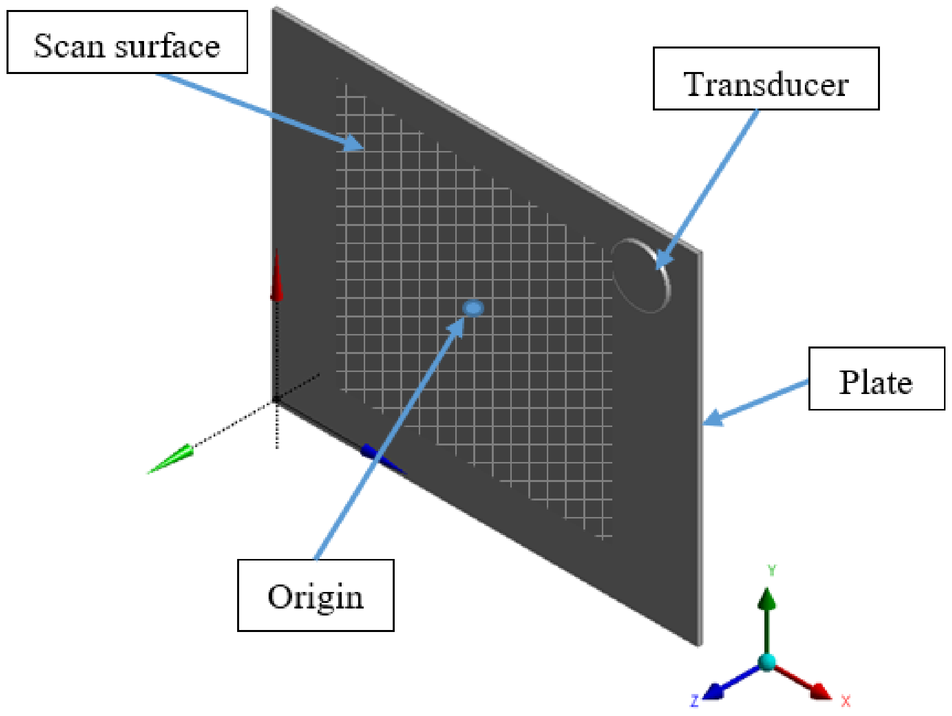


Figure 3: Example plate geometry with transducer on plate's surface

After completing this task, the reader should have geometry *.step* files for each desired model setup created during the simulation scheme. These geometry files should be located in the *geometry* folder.

3.4.2 Setting up the ANSYS FEA solver

Now that all plate geometry files are created, the steady-state ultrasonic excitation response of the plate can be obtained by a harmonic response simulation in ANSYS Mechanical. The procedure for setting up this simulation follows previous work by O'Dowd et al. [1] The simulation setup procedures are included in the *ansys_sim_tutorial.docx* file in the documentation folder. Any of the *.step* files generated in the previous section can be loaded in for the geometry instead of recreating the geometry as indicated in the simulation procedures. The ANSYS generic Aluminum Alloy material was used for this study. The procedures include the ADPL commands that must be inserted into ANSYS Mechanical to export the steady-state response data of the plate. Once the ANSYS setup procedures are completed, save the workbench project in the *ansys* directory under the project root folder.

3.4.3 Automating ANSYS simulations

Once the simulation has been setup in ANSYS and saved, it is now ready to load the geometry *.step* files, run the harmonic response simulation, and export the steady-state response data files. Depending on the platform and system, these tasks take several minutes to complete per geometry file. For this study, this workflow was automated to sequentially load geometry files into the ANSYS workbench project, run the harmonic response simulation, export the data, and optionally run the data through Matlab image processing and offload to Dropbox cloud storage; the *run_sims.py* script performs these tasks programmatically. The reader must make a few edits to this script file before running:

1. Change the SetUserPathRoot() DirectoryPath argument to the absolute path to your project root folder on your system (i.e. C:\Users\eckelsjd\DeepWaves).

2. (Optional) Install Matlab engine for Python and uncomment the two Python os system calls in the main *for* loop which execute the *run_matlab.py* Python script.

3. (Optional) Install the Matlab statistics toolbox and edit the *cmd* variable from the previous step to specify what kind of segmentation mask you would like to automatically generate. See the Matlab *plot_wavefield.m* source code file for options. This is the Matlab function called from within Python via the arguments specified in the *cmd* variable. Also reference the *exec_matlab()* function in the *run_matlab.py* source code file for more information.

4. (Optional) Install the Dropbox Python API, setup a Dropbox account, and copy the Python API account key into the *run_matlab.py* script. Follow online tutorials and the inline comments in the script to adjust the *upload* function so that it properly uploads files to your Dropbox account. Uncomment the calls to the *upload* function in main() to move your files off the hard drive to your Dropbox account. Note: you should feel comfortable reading through the *run_matlab.py* source code and making edits so that the Dropbox API can properly find your account and move all associated files.

5. Open ANSYS workbench. Go to File→scripting→Run Script file and select *run_sims.py*. The script will begin loading all *.step* files from the geometry folder and running the simulation in ANSYS mechanical. You will see data .txt files show up in the data folder as simulations complete.

6. If you opted to run Matlab image processing and/or Dropbox file offloading, you will see a command window pop up and indicate its progress, and image files (and segmentation masks,

if indicated) will begin populating the images and labels folders, respectively. All errors that occur while scripting will be written to the *run_sims.log* file in the logs folder, and you may also see errors appear in the ANSYS workbench GUI. Errors that occur during Matlab image processing or Dropbox file handling are harder to catch; these errors will cause the command window to close abruptly and ANSYS workbench to stop executing. If this happens, you will need to test the *run_matlab.py* file separately from ANSYS workbench, perhaps in a Windows command prompt running interactive Python. Errors with Matlab are likely due to incorrect arguments passed to *plot_wavefield.m.* Errors with Dropbox are likely due to incorrect file handling or account authentication errors (incorrect keys, IDs, namespaces, etc.).

3.5 Image processing

The default output of the ANSYS automation performed in the previous section is a *real.txt* and an *imaginary.txt* file for each simulation performed, representing the real and imaginary deformation response data, respectively. These data files will be located in the data folder, and will be renamed based on the filename of the *.step* files. For example, *geometry1.step* will produce two data files in the data folder: *geometry1_real.txt* and *geometry1_imaginary.txt.* These filenames can be passed directly to the Matlab *plot_wavefield.m* function, along with a few other parameters, to generate the wavefield images for the data (real, imaginary, and magnitude), and optionally create a segmentation mask image. This function can also optionally export a complex-valued deformation matrix .mat file by uncommenting the appropriate lines in the source code. These complex matrices are used in AWS processing to generate AWS wavenumber map predictions. All generated image files are saved to the images folder; all segmentation masks are saved to the labels folder; all .mat files are saved to the mat folder. See the source code of *plot_wavefield.m* for more details.

There are three ways to perform the Matlab image processing:

1. Call *plot_wavefield.m* on one set of data directly. This is mostly useful for debugging and checking output of a single simulation.

2. Run the *process_images.m* Matlab script. This simply calls *plot_wavefield.m* in a *for* loop; make changes to the *plot_wavefield.m* function call within *process_images.m* to change the arguments. The script will generate wavefield images for all data files in the data folder automatically; this requires all complementary real.txt and imaginary.txt files to be present in the data folder at the time of execution. This script allows you the option to perform batch normalization over the entire dataset before generating the wavefield images as well. In order to use the *parfor* loop in the script, you must have the Matlab Parallel Computing toolbox; if you don't, just change the *parfor* loop to a normal *for* loop (with slower run times).

3. Make the optional changes listed in the previous section to automatically generate wavefield images from within the ANSYS Python script directly.

The wavefield images generated during image processing are used as the raw input to the convolutional neural network (CNN) for training. In this study, only the real surface response wavefield images were used for CNN training. A segmentation mask "label" was generated for each real wavefield by specifying the "default" argument to the *plot_wavefield.m* function. The ground truth segmentation mask was generated by specifying the "color" argument instead (only for visualization and comparison). The default segmentation label images (in the labels folder) appear completely black. Each pixel is given a numeric value equivalent to its class identifier. In this study, there were 10 plate thickness classes to identify in each image on a pixel by pixel basis, as summarized in table 3.

Table 3: 10 thickness classes with numeric class identifiers

Thickness class	**Class identifier (pixel value)**
10 mm	0
9 mm	1
8 mm	2
7 mm	3
6 mm	4
5 mm	5
4 mm	6
3 mm	7
2 mm	8
1 mm	9

In a given 10 mm thick plate with no defects, every single pixel value in the segmentation label image would have an RGB value of $(0, 0, 0)$, equivalent to its class identifier. If a corrosive defect was introduced to the back of the plate, causing the thickness of the plate in that region to be reduced to 7 mm, then all pixel values in that region of the segmentation label image would have an RGB value of $(3, 3, 3)$. Thus, the CNN will be given wavefield images of a plate and it will be told (via the label images) which wave patterns and regions in the image correspond to what plate thickness value. In this way, the CNN will be trained to classify local plate thickness based on local wave patterns and features, irrespective of the wave modes present.

Segmentation mask limitations

There are several limiting cases that cause problems with the current automatic segmentation mask generation. The segmentation mask section of the *plot_wavefield.m* file has several comments on how the code works. In general, 3D mesh node point coordinates (X,Y,Z) are grouped according to their Z-value. Nodes on the surface of the plate have a Z-value equal to the plate thickness. The filename of the data specifies the thickness of the plate at the surface, as well as at the defect location. From the filename alone, if you have a 10mm thick plate with a defect of thickness 4mm, the code will parse this information out. At the defect location, there will be a flat surface at a Z-value equal to the plate thickness minus the thickness at the defect (a Z-value of 6mm in this case). Using the *kmeans* function in the Matlab Statistics toolbox, the node points at this Z-value are grouped into a cluster, and all node points at this Z-value that are far away from the cluster are filtered out. Now we have obtained the location and depth of the defect from the filename alone. However, the mesh node (X,Y,Z) coordinates are not in line with the uniform grid of the wavefield image; we must map these random mesh points to the uniform grid. This mapping is done using the Matlab built-in function *dsearchn*. This function takes all of the mesh node coordinates and maps them to the closest points on the uniform grid. However, the mapping is rather sparse because the uniform grid is much more refined than the mesh node points. After a series of blurring, edge detection, and image hole-filling, the uniform grid is a segmentation mask that is white at the defect and black everywhere else. We can create a segmentation mask like this for every thickness class, set the white spots equal to the appropriate class identifier, and logically AND each thickness segmentation mask together to combine into one segmentation label image for the CNN. This same process is done for the ground truth images, just with 3 channels for RGB visualization.

Here are the current limitations of this approach:

1. Must have the Matlab Statistics toolbox.

2. Increased processing time.

3. Filename must follow conventions for the segmentation masks to generate properly. For each new round, you must give some indication of the plate thickness and the number and thickness of defects. Future work would eliminate these dependencies somehow.

4. The *kmeans* function filters only 1 cluster by default. This method only supports segmenting when there is exactly one defect at a given thickness, not for example when there are two 1mm defects in different locations on the plate. To fix this, you would need some way for Matlab to automatically detect how many clusters there are, or to include how many clusters there are in the filename so you can pass a cluster argument to *kmeans*.

5. Performance is only reliably good when there is only one defect total present in the plate at any thickness. When there is more than one defect, the mesh node points often cluster in locations other than the surface of the defect, which throws off the *kmeans* algorithm for filtering out only the defect's mesh node points. There are cases where there might be a 1mm defect and a 5mm defect somewhere else in the same plate, and the segmentation mask is still generated just fine. This just needs more testing and debugging to figure out the real problem behind this limitation. One possible solution is to only export ANSYS mesh node point data at select regions or Z-values rather than having Matlab do all of the filtering.

6. Along with the previous limitation, overlapping defects often overwrite each other and produce incorrect segmentation masks. The order in which the logical ANDs are performed to combine different thickness masks affects the performance on this problem.

If the reader comes across a case where the segmentation mask is not generating properly, then it most likely falls in one of the categories above. The reader should always visually check the generated masks to make sure the code is performing correctly. If there is no way to correctly generate the mask using this code, then Matlab can be used to hardcode the location and depth of the defects and draw colored shapes on a uniform grid. However, this method would be rather difficult with irregular shapes. It's best to stick to regular shapes, like circles and squares, if you would prefer to make segmentation masks this way. It is also possible, albeit rather tedious, to use Matlab's image processing tool, or some other labeling software to generate segmentation masks by hand. Despite the automatic mask generation's limitations, it is still much more adaptable to changing geometry. Future work would improve upon the limitations listed above.

3.6 Data augmentation

After generating all wavefield images and segmentation labels, the overall dataset size should be increased by augmentation. Several common data augmentation techniques, such as blurring, warping, lighting, and tilting would cause physical wave pattern features to change in the wavefield images in an undesirable way. In this study, four data augmentation techniques were identified to avoid these warping effects: adding Gaussian noise, rotating 180 degrees, vertical flipping, and horizontal flipping. All plate geometries were designed to only have defects in the upper right quadrant of the plate. A 180 degree rotation was applied to every image to double the dataset size. Then, Gaussian noise was added to every image in the new dataset to double in size again. An original dataset size of 200 images would increase to 800 images, while still only having to perform 200 simulations. During CNN training, horizontal and vertical flipping were applied randomly to the dataset between training epochs. In this way, the CNN was exposed to defects in all four quadrants of the plate. The *augment.py* Python script uses the *numpy* and *scikit-learn* Python libraries to perform the data augmentation described here. All new augmented images were saved in the images folder with new filenames. Rotations were also applied to segmentation labels and saved back into the labels folder.

3.7 CNN training

The completed raw dataset consisted of all original real wavefield images and augmentations in the images folder along with all corresponding segmentation label images in the labels folder. The dataset was uploaded to a Google Cloud Platform (GCP) virtual machine (VM) with GPU-acceleration capability. One of the source code Jupyter notebooks were opened on the VM and accessed by static IP addressing from the host machine. All Fast.ai CNN training procedures were followed cell by cell in the Jupyter notebook, including loading the dataset and labels, splitting the dataset 20/80 into validation and training sets, loading a U-net CNN architecture with a pre-trained ResNet encoder, searching and selecting an optimal learning rate, and training the CNN on the dataset for several epochs. The CNN must be able to find corresponding label images based on the filenames of the wavefield dataset images. For enhanced training, the CNN was saved to *.pth* files and loaded later after changing several training parameters.

See the *training_instructions* files in the documentation directory for more details on performing the CNN training. The two training instructions documents correspond to two different methods performed in this study for CNN training. The procedures described in this document and in *training_instructions_JE.txt* are to be used with the *unet_train_JE.ipynb* Jupyter notebook. The *training_instructions_KH.docx* file outlines procedures to use the *unet_train_KH.ipynb* Jupyter notebooks. The *source_files.txt* document also provides more detail on using all source files mentioned in this section.

3.8 Output

After training, the CNN was exported to a *.pkl* file for deployment on another system. The CNN model displayed its intersection-over-union (IoU) value and F1 score (dice coefficient) during training to indicate its accuracy on the validation set. The model's performance on the validation set was also quantified by a confusion matrix on each class value. Test images were generated similar to the original dataset and were passed to the trained CNN for inference prediction. The CNN outputted a thickness prediction map in a linear-thickness color scale, indicating the regions of the plate with different thickness values. An example of this output is shown in figure 1.

4 Future work

After reaching the end of the previous section, you have made it as far as we did at the time of writing. This section will speculate on things we wish we had time for, or places we believe the project could go in the future.

4.1 Addressing current limitations

Several limitations have been discussed throughout this document with respect to software bugs and hardware limitations. Areas of special interest that need more work are generating segmentation masks automatically from ANSYS data, thinning the data exported from ANSYS to decrease storage requirements, and searching for alternatives to ANSYS simulations to speed up simulation processing time (or just throw more GPUs at it).

4.2 Comprehensive dataset

In this study, a total of 8 different rounds of simulations were performed to account for as many physical scenarios as possible. This effort was extensive, but certainly not exhaustive. There are several real-world experimental test cases that fall outside of the parameters our simulations accounted for. To improve the robustness of the CNN model, the training dataset must be larger and more comprehensive. Some cases to consider include changing plate dimensions, variable transducer frequencies, multiple transducers on the same plate, non-plate-like structures, training on experimental data in addition to simulated data, structures where higher wave modes (other than A0 and S0) are present, thicker plates, unbounded plates, plates with multiple defects in multiple orientations and dimensions, etc.

4.3 Regression

The task at hand is an image segmentation problem, which is inherently a classification problem, just classification on the pixel level. The authors of the current study started here as a proof-of-concept. Ultimately, however, the problem at hand of identifying defects in a plate is a regression problem, where defects are allowed to take on any value in a continuous range, not just one of a set few discrete values. It is not currently known if there are existing CNN models out in the field that perform regression on an image segmentation task. Perhaps further computer science research into the data being passed to the fully-connected layers at the end of the image segmentation CNN could indicate a way to output classifications on a continuous scale rather than a discrete one. This would be an excellent area of further research, and is most recommended by the authors.

4.4 CNN model refinement

None of the authors have great experience in the field of machine learning and neural networks. The Fast.ai library was chosen as a machine learning platform because of its short on-ramp in generating decent results with little prior training. However, Fast.ai hides most of the inner-workings of the network (intentionally) from the user. Further optimizations of the network would require more training and a shift away from Fast.ai into a more mature platform like TensorFlow or PyTorch, where customization is more open and achievable. Once this shift is made, more state-of-the-art models, like the recent Refine-net image segmentation model [2], can be implemented in code and trained on our dataset for improved performance.

4.5 CNN training refinement

In conjunction with the previous statement on CNN model refinement, moving to a more mature platform would also allow further testing and refinement on training hyperparameters, such as weight decay, batch size, and learning rate to improve model accuracy. Most of these hyperparameters were left default in this study, but future work should experiment with these values and find the optimal parameters for training the CNN model.

References

[1] N. M. O'Dowd, D.-H. Han, L.-H. Kang, and E. B. Flynn, "Exploring the performance limits of full-field acoustic wavenumber spectroscopy techniques for damage detection through numerical simulation," *Proc. 8th EWSHM*, 2016.

[2] G. Lin, F. Liu, A. Milan, C. Shen, and I. Reid, "Refinenet: Multi-path refinement networks for dense prediction," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 5, pp. 1228–1242, 2020.