ARL-TR-8372 ● Jun 2018

US Army Research Laboratory

Convolutional Neural Networks for 1-D Many-Channel Data

by John S Hyatt, Eliseo Iglesias, and Michael Lee

Approved for public release; distribution is unlimited.

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-8372 ● Jun 2018

US Army Research Laboratory

Convolutional Neural Networks for 1-D Many-Channel Data

by John S Hyatt and Michael Lee
Computational and Information Sciences Directorate, ARL

Eliseo Iglesias
Vehicle Technology Directorate, ARL

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
June 2018	Technical Report	September 2017–May 2018

4. TITLE AND SUBTITLE
Convolutional Neural Networks for 1-D Many-Channel Data

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
John S Hyatt, Eliseo Iglesias, and Michael Lee

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
ATTN: RDRL-CIH-C
Aberdeen Proving Ground, MD 21005

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-8372

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Deep convolutional neural networks (CNNs) represent the state of the art in image recognition. The same properties that led to their success in that domain allow them to be applied to superficially very different problems with minimal modification. In this work, we have modified a simple CNN, originally written to classify digits in the MNIST database (28 × 28 pixels, 1 channel), for use on 1-D acoustic data taken from experiments focused on crack detection (8,000 data points, 72 channels). Though the model's predictive ability is limited to fitting the trend, its partial success suggests that the application of convolutional networks to novel domains deserves further attention.

15. SUBJECT TERMS
machine learning, regression, ultrasound, material fatigue, nondestructive testing

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
					Michael S Lee
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER (Include area code)
Unclassified	Unclassified	Unclassified	UU	44	(410) 278-5888

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures iv

List of Tables iv

Acknowledgments v

1. Introduction 1

2. Methods: Adapting an Image Classifier to 1-D Signal Regression 2

2.1 Crack Detection via Acoustic Pitch/Catch Sensors 3

2.2 MNIST Classifier to Signal Regressor in Four Steps 5

2.3 Further Enhancements 7

2.3.1 Data Preprocessing 8

2.3.2 Dropout 9

2.3.3 *k*-fold Cross-validation 10

2.3.4 Hyperparameter Optimization 10

3. Results and Discussion 12

4. Conclusions 14

5. References 16

Appendix. Python Scripts for a 72-Channel Acoustic Signal Regressor 17

List of Symbols, Abbreviations, and Acronyms 36

Distribution List 37

Approved for public release; distribution is unlimited.

List of Figures

Fig. 1 A schematic of the experimental setup is shown in panel a. Oscillatory extensile stress is applied to the metal plate via clamps attached to its wide ends. Panels b–d show a close-up of the crack propagation area over the course of one experiment. The square markings are 1 mm on a side. 4

Fig. 2 A schematic of differences between the original MNIST classifier and our regressor. There may be more than three convolution/pooling layer pairs in the models tested in this work. 7

Fig. 3 Representative acoustic measurements representing the lowest and highest of the eight measured frequencies. The first row shows the "pitch" signal, and the other rows show the "catch" signals received by the sensors. Rows 2, 3, and 4 show paths 1–6, 2–5, and 3–4, respectively. Blue lines represent the baseline taken t = 500 s into the fatiguing process. Orange lines are from measurements taken just before complete failure, t = 27,000 s. 8

Fig. 4 Predicted vs. actual t/t_c for the 14 cross-validation data sets. The black line has a slope of 1. 13

Fig. 5 Predicted vs. actual t/t_c for the two test data sets. The black line has a slope of 1. 14

List of Tables

Table 1 Hyperparameter ranges 11

Table 2 Optimized hyperparameter values 12

Approved for public release; distribution is unlimited.

Acknowledgments

We would like to thank M Coatney, A Hall, R Haynes, and R Valisetty for helpful discussions. Computer time was provided by the US Army Research Laboratory's Department of Defense Supercomputing Resource Center.

Approved for public release; distribution is unlimited.

1. Introduction

Machine learning, in a nutshell, is the process by which an algorithm builds a model based on a certain amount of example data and uses that model to make predictions, given new data. In general, the specifics of the model are not set by the programmer. Rather, the programmer provides an outline of the model's architecture along with example data. The training algorithm then iteratively searches within that framework for the model that best describes the example data.

Of the many possible architectures, we focus on deep convolutional neural networks (CNNs).[1] Though there are many variations on the theme, standard CNNs are generally built from the same basic components:

- **Input layer** – The input data, often a collection of images or time-series (audio, video) data of fixed size/length over all input samples.
- **Convolutional layer** – A collection of filters, much smaller than the input (e.g., 5 × 5 pixels, with the same depth as the input). Each of these filters is convolved with every patch of the same size in the original image. The output depth (i.e., number of output channels) is equal to the number of filters.
- **Pooling layer** – Usually a 2 × 2 max pooling layer, in which a 2 × 2 patch of pixels is represented by the highest-valued pixel in that patch. This downsamples the data in the previous layer by a factor of 4. Convolutional and pooling layers are usually alternated several times, with the output of each pooling layer serving as the input of the next convolutional layer.
- **Dense layer** – After the final pooling layer, a dense layer gives the output of the network (e.g., with one node for every class in a classifier). The many-channel output of the convolutional and pooling layers must be flattened (number of dimensions reduced to one) before being passed to the dense layer.

Because the filters in a convolutional layer are applied identically to each patch of input pixels, it can recognize features larger than one pixel and requires many fewer parameters to describe than does a dense layer capable of processing the same input.

The filters of a trained network encode the features it has learned, and which it looks for when presented with new data. Because the pooling layers repeatedly downsample the data as it propagates through the network, each successive convolutional layer looks at a larger fraction of the original input image. Moreover, the output of each convolutional layer (an "image" of filter activations, with one

Approved for public release; distribution is unlimited.

channel per filter) is fed to the next-deeper layer as input. Thus, the filters in deeper layers correspond to features that are not only larger, but more complex as well.

None of this implies that CNNs are restricted to 2-D inputs. In fact, while a black-and-white image has only two dimensions (height and width, in pixels), a color image has three dimensions: height, width, and depth (red, green, and blue [RGB] values separated into three channels). Image analysis applications routinely accommodate this higher dimensionality not by combining the color channels during preprocessing, but by using convolutional filters whose dimensions match those of the input layer.

Similarly, 1-D data can be modeled by a CNN with 1-D kernels, and the resulting model can accommodate an arbitrary number of channels. The idea that CNNs can be used to model time-series data is not new.[2] However, despite the fact that signal analysis is key to many applications in engineering, medicine, and other fields, we know of very few occasions where CNNs have been adapted to this domain.[3,4] For the most part, they have been restricted to image analysis applications, although they have been used in speech recognition and natural language processing as well.[5] None of these applications combine data from a large number of channels.

Our primary goal is to illustrate the potential usefulness of CNNs in modeling 1-D, many-channel data via application to a real-world problem. Despite using a very simple CNN to regress quite noisy data, we show that it is possible to obtain meaningful results. Additionally, while we assume basic familiarity with machine learning concepts, we have deliberately shifted our focus away from the technical details as much as possible (beyond what is necessary to explain our work). This is because, as a secondary goal, we want to provide a working example of practical CNN implementation, accessible to as broad an audience as possible. Machine learning is a fast-growing field, and many useful tools are new or in a state of active development; as a result, there are not many such examples that include these tools.

2. Methods: Adapting an Image Classifier to 1-D Signal Regression

Somewhat counterintuitively, there is little fundamental difference between a CNN designed to classify images and one that interprets 1-D signals. This is because, in both cases, all the CNN "sees" is an array of numbers that represent either pixel intensity at a particular coordinate, or signal amplitude as a function of time.

In this section, we discuss a 1-D signal regression problem, namely using acoustic signals at discrete frequencies to monitor crack formation in a fatigued metal plate. We then describe the fairly small alterations that must be made to a simple image

Approved for public release; distribution is unlimited.

classifier to produce a regression algorithm ("regressor") suitable for examining this problem. Finally, we combine several existing machine learning training and optimization tools to develop the best possible signal regressor within the constraints of the original simple classifier's architecture.

2.1 Crack Detection via Acoustic Pitch/Catch Sensors

Condition-based maintenance is the concept that equipment maintenance is only performed when necessary, rather than at fixed intervals. It is motivated by the prospects of cost savings, increased efficiency, and decreased maintenance-related downtime, but is not easy to implement. The chief reason for this is that every piece of equipment must be monitored in real time to identify deteriorating components to be replaced before they become dangerous.

We have received data from our colleagues in the Vehicle Technology Directorate of the US Army Research Laboratory. The precise experimental setup is described elsewhere.[6] The goal of their project was to find a way to quantify the condition of a fatigued metal plate by using acoustic measurements to answer questions like, "Is the plate about to crack? Has it cracked? How much has the crack grown?"

In those experiments, aluminum plates are prepared as shown in Fig. 1. A hole is drilled through the center of the plate, with a notch cut into one side to promote crack formation. Three piezoelectric actuators are attached to one end of the plate, and three sensors to the other end. After a baseline measurement, the plate is fatigued (subjected to oscillatory extensile stress), with acoustic measurements made every 500 s.

During the measurement intervals, the stress is relaxed, and acoustic signals, emitted by the actuators, are recorded by the sensors after passing through the plate.

Because there are three actuators (labeled 1–3 in panel a of Fig. 1) and three sensors (4–6), the acoustic signals follow nine unique paths, each including the summed contributions of direct transmission and acoustic reflections. Only one actuator is active at one time. Moreover, each actuator successively emits signals in eight distinct frequencies, ranging from 150 to 500 kHz in increments of 50 kHz. The sensors thus record 72 data sets (signal amplitude vs. time), each containing 8,000 data points (sampling frequency 48 MHz).

Approved for public release; distribution is unlimited.

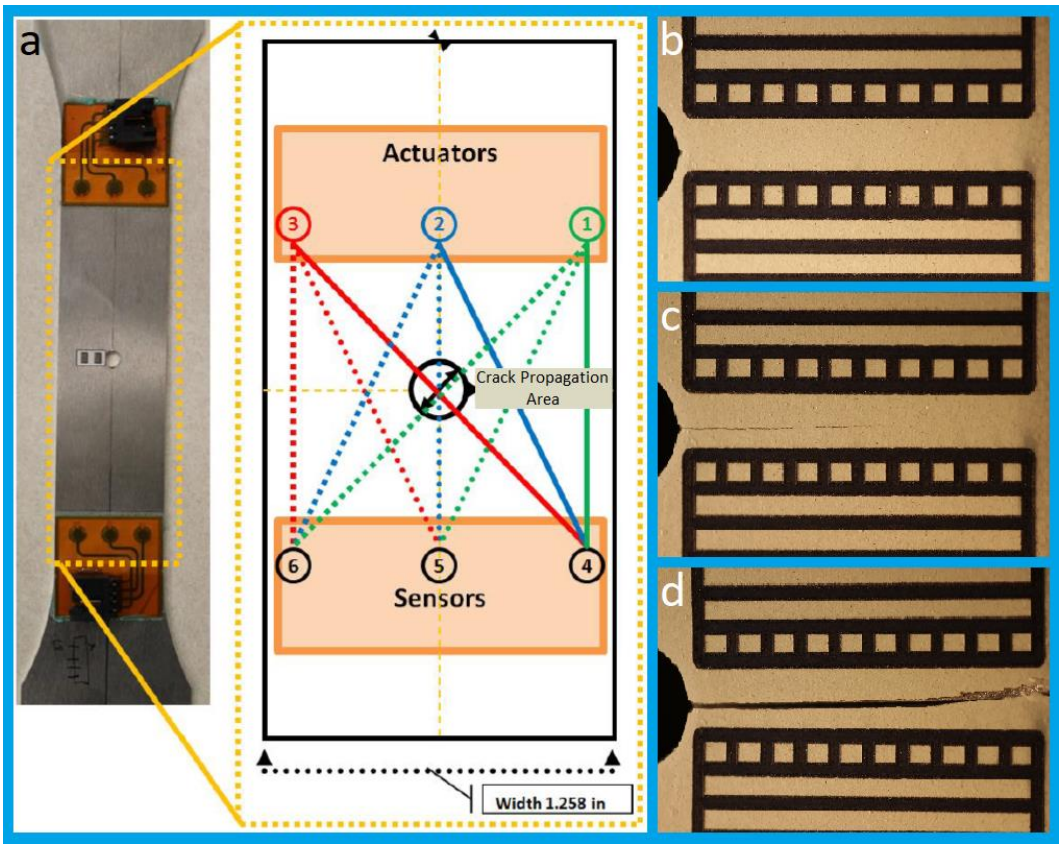

Fig. 1 A schematic of the experimental setup is shown in panel a.[6] Oscillatory extensile stress is applied to the metal plate via clamps attached to its wide ends. Panels b–d show a close-up of the crack propagation area over the course of one experiment. The square markings are 1 mm on a side.

Changes in the plate's microstructure due to ongoing fatigue are much too small to affect the acoustic signals, which initially remain unchanged from one measurement to the next. Eventually, however, a crack forms at the notch and begins to propagate through the shaded region marked "Crack Propagation Area" in panel a of Fig. 1.

Panels b, c, and d of Fig. 1 show a close-up of this region, photographed over the course of the experiment. In panel b, the experiment has just begun and there is no damage to the plate. Panel c shows the plate after a fatigue-induced crack has propagated about halfway to the edge of the plate, and panel d shows the plate at the end of the experiment, after it has cracked all the way through.

The crack propagation area cuts directly through only three of the linear paths: 1–4, 1–5, and 2–4. However, the crack's presence affects the signals received along every path, since each signal incorporates reflections from the boundaries of the plate, and the growing crack changes those boundaries. The problem thus involves obtaining a single measure of "condition" from 72 channels of 1-D data. Because

Approved for public release; distribution is unlimited.

the acoustic measurements are not sensitive to changes in the material before a crack forms, we chose "time since a crack has formed" as our condition indicator, since we know the crack will continue to grow with time. We defined t_c as the time a crack is first visible (by comparing photographs taken at successive measurements with the one taken at the beginning of the experiment), and our condition indicator as t/t_c. When $t/t_c < 1$, no crack has visibly formed. Larger t/t_c corresponds to a longer crack and therefore more fatigue damage.

On the surface, this type of regression problem has little in common with image classification. A regression algorithm has only one output (in this case, the condition indicator of the plate), which is generally a continuous value. A classifier, on the other hand, sorts input data into one of several discrete categories.

However, the entire architectural difference between two CNNs—one a regressor, the other a classifier—lies in the final layer of the network, specifically the output shape and the activation function, if any. Similarly, the entire architectural difference between CNNs that accept input data sets with different dimensionalities is contained in the shape of the convolutional filters and pooling layers.

Concerning the data itself, the 1-D, 72-channel acoustic data described above are distinguished from 2-D, 3-channel RGB images only by the shape of the arrays needed to contain them. In the same way that multispectral imaging combines light intensity from multiple spectral bands, this data combines the acoustic intensity from multiple frequencies of soundwaves. Taking all of this into account, it makes sense to look for a CNN that can regress the condition indicator of the metal plate at some time from the acoustic signals measured at that time.

In the remainder of this section we discuss the original Modified National Institute of Standards and Technology (MNIST) classifier and the ways it is changed to obtain our condition regressor.

2.2 MNIST Classifier to Signal Regressor in Four Steps

The MNIST database is a collection of 70,000 handwritten digits (the numbers 0–9), formatted as 28 × 28 pixel grayscale images. These are divided into a training set (60,000 images) and a test set (10,000 images). CNNs gained widespread recognition after one was used to correctly sort these digits into the correct categories[7] for use by banks and the United States Postal Service. By many standards, the problem is quite forgiving, and partly for this reason it remains a popular first test case for new classifiers (and new CNN practitioners).

An MNIST classifier represents one of the most basic CNNs. We want to keep things as simple as possible and highlight the similarities between a 2-D image

Approved for public release; distribution is unlimited.

classifier and our 1-D signal regressor. In line with those goals, we begin with an MNIST classifier and convert it to a regressor in four steps.

The initial classifier has three 2-D convolutional layers with 8, 16, and 24 kernels, respectively. Each kernel is 3 × 3 pixels (stride 1), and each convolutional layer is followed by a max pooling layer with pool size 2 × 2. The final two layers are a global average pooling layer and a dense layer with 10 nodes, which represent the 10 possible output categories.

An MNIST classifier can reach high validation accuracy with fewer layers and/or kernels, but takes more epochs to train. This architecture represents a good balance between simplicity and training speed (this model trains to less than 1% validation error in fewer than 20 epochs). Similarly, the kernel and pool sizes in each layer do not have to be fixed at 3 × 3 and 2 × 2, respectively, but those values, particularly for the pool size, are customary and work well.

To convert this simple image classifier to a regressor suitable for our problem, we only needed to make four changes, illustrated in Fig. 2:

- **Change the input shape** – The MNIST images are 28 × 28 pixels and only have one channel, so the input data for the classifier has the shape `(28,28,1)`. Each data set for our signal regressor has 8,000 time steps and (9 paths) × (8 frequencies) = 72 channels, so the input data for the regressor has the shape `(8000,72)`.
- **Change the dimensionality of the convolution and pooling layers** – Keras has separately defined layers for 1-D and 2-D inputs (because the convolution and pooling layers treat each channel separately, they are not counted as "dimensions" for this purpose). Simply change the `Conv2D`, `MaxPooling2D`, and `GlobalAveragePooling2D` layers to `Conv1D`, `MaxPooling1D`, and `GlobalAveragePooling1D`.
- **Change the final dense layer** – The classifier's final dense layer has 10 nodes and a softmax activation function,[8] which ensures that the sum of the 10 values produced by a given input image adds up to 1, each of the 10 giving the probability that the input corresponds to a particular digit. Our signal regressor has only one output, namely the condition indicator (the time since a crack has formed); therefore, it needs only one node. Because the condition indicator does not have an upper bound, we also remove the softmax activation function; thus, we change the final layer from `Dense(10, activation='softmax')` to `Dense(1)`.

Approved for public release; distribution is unlimited.

- **Change the loss function** – The loss function to be minimized during training represents the "distance" between the values outputted by our model, given some input data, and the "true" values. In the case of the classifier, this is the cross entropy between the probability distribution predicted by the classifier, and the true probability distribution given by each image's label. Our regressor uses mean squared error, so we changed the loss function from `loss='categorical_crossentropy'` to `loss='mean_squared_error'`.

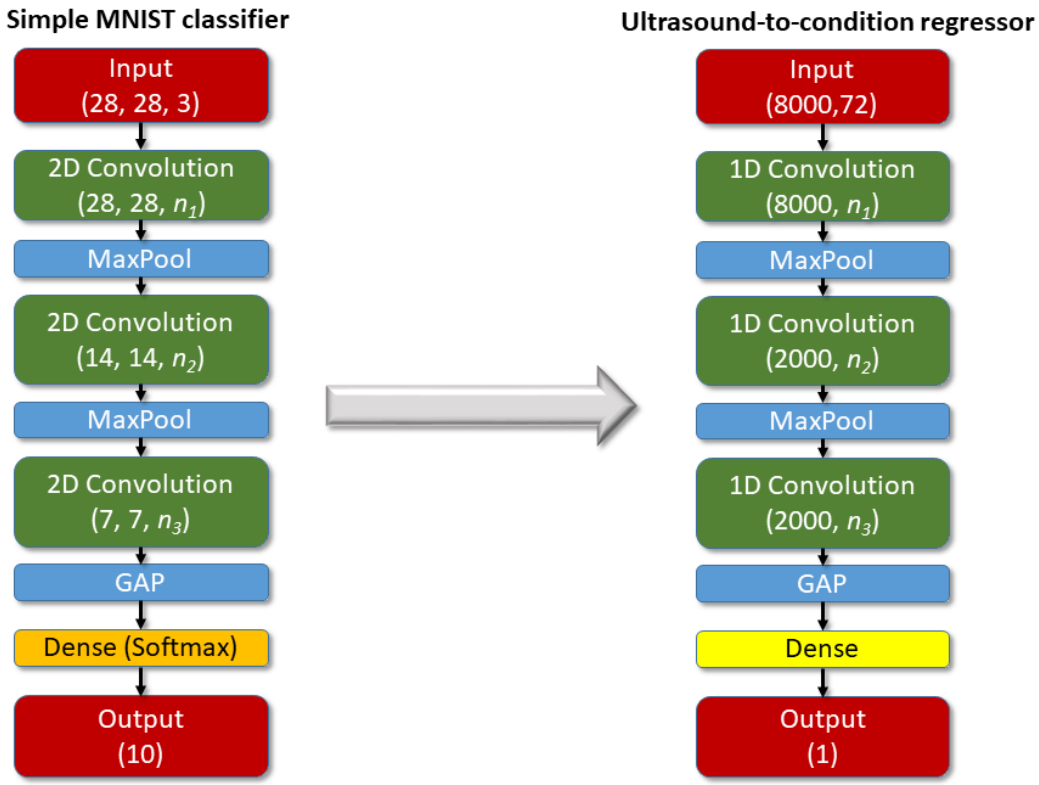

Fig. 2 A schematic of differences between the original MNIST classifier and our regressor. There may be more than three convolution/pooling layer pairs in the models tested in this work.

2.3 Further Enhancements

With these changes, the model is now capable of receiving the acoustic signals as input and producing the right type of the output. However, we can make the model somewhat more flexible and robust by adding several additional features, described below. The code for our final regressor, including these features, is shared in the Appendix.

2.3.1 Data Preprocessing

The acoustic data (actually voltage generated by the piezoelectric sensor in response to acoustic oscillations) were taken at snapshots every 500 s during the fatiguing process. Each data set contains 72 individual measurements (one for each combination of nine paths and eight frequencies), and each of those contains 8,000 data points.

Figure 3 shows representative raw measurement data for several frequencies, paths, and measurement times. The "pitch" signal, shown in black and echoed in the data, dominates the early signal, even when the direct path between actuator and sensor is broken by a discontinuity in the metal. However, as the figure shows, the later parts of the signal vary substantially for a given frequency both across different paths and over time within the same path.

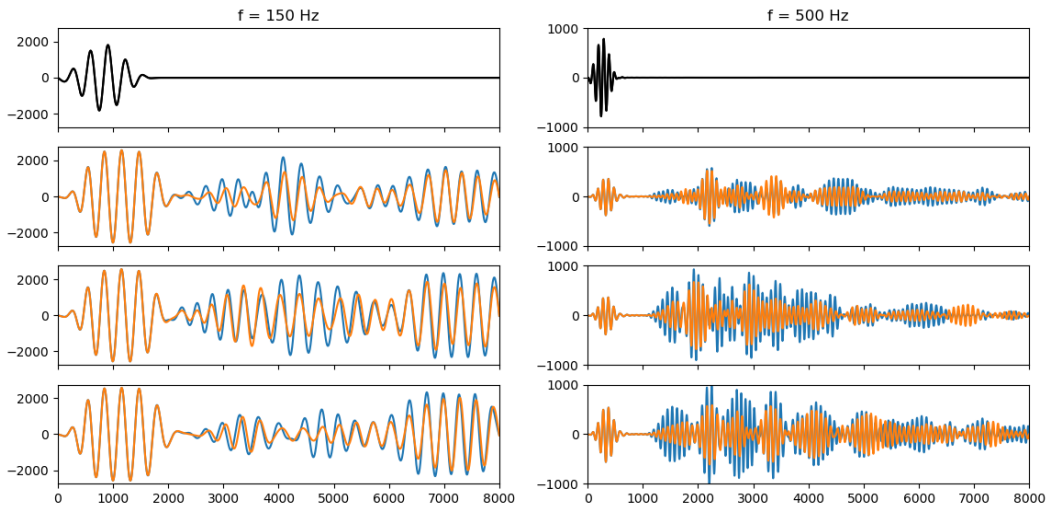

Fig. 3 Representative acoustic measurements representing the lowest and highest of the eight measured frequencies. The first row shows the "pitch" signal, and the other rows show the "catch" signals received by the sensors. Rows 2, 3, and 4 show paths 1–6, 2–5, and 3–4, respectively. Blue lines represent the baseline taken *t* = 500 s into the fatiguing process. Orange lines are from measurements taken just before complete failure, *t* = 27,000 s.

Particularly for the lower frequencies, the measurement window is not long enough to capture a significant portion of the true response, which is cut off before it has fully decayed. Unfortunately, this likely means any model we develop will not benefit from training on any information contained in this low-frequency, long-time region.

Also, note the different *y*-scales used for the two frequencies. Because the amplitude of the signal varies, we rescaled the data as part of a preprocessing step. Each channel is normalized by the standard deviation of that channel across all measurements, including those from different experiments. It should be noted that

we normalized the training and validation data sets separately. The data is oscillatory and therefore already very close to zero-mean.

Furthermore, because experiments differed slightly in the amount of time it took the plate to crack completely, some experiments had more data sets than others. Our Python code relies on the data being collected into arrays. Because the dimensions of an array must be internally consistent, we randomly removed points from the longer experiments until all had the same number. Matching the number of trained samples per experiment also balances the data to ensure some experiments are not more impactful than others.

Finally, the experiments also varied in the amount of time before the plate began to crack. Since the acoustic signal is not sensitive to changes in the plate microstructure before a crack forms, the data has no built-in way to account for this variability. In an attempt to do so, we scaled the experiment duration, t, by the "crack time," t_c, the time at which a crack became visible. We were provided with photographs taken of the Crack Propagation Area at each measurement interval for this purpose. As shown in Section 3, scaling based on the photographs was not perfectly accurate in every case, but close enough to be helpful.

2.3.2 Dropout

Generally, the performance of a CNN increases with the depth and complexity of its architecture, as it can then learn more complicated relationships between its input and output.[9] An important caveat to this is that a more complex model is also more likely to overfit the data it is trained on, and not generalize well to new data. A training procedure called "dropout" is one way to resolve this dilemma, allowing the network to contain a high degree of complexity while avoiding overfitting.

The concept of dropout was originally developed to address this problem in dense neural networks (not CNNs).[10] The idea is that, during training, each node has a probability d of being dropped, or temporarily removed, from the network for the current epoch. Then, in the final trained model, all nodes are included and their weights are scaled by d. The result is an approximate average of the predicted outputs of many networks, each with a different architecture, all sharing the same weights. This both forces the model to incorporate redundancy, and guards against overfitting.

Note that while dropout can be used while training a CNN (individual filters have probability d of being dropped out in a given epoch), convolutional layers already have fewer parameters than an equivalent dense layer by design. (This is actually the whole point of convolutional layers in the first place.) This means that dropout has less of an effect when applied to a convolutional layer, because there are fewer

Approved for public release; distribution is unlimited.

parameters to be dropped, but it still has a net positive effect on the CNN's performance.[10]

In our model, we incorporate dropout after each convolution/max pooling layer pair, using a single value of *d* across all layers.

2.3.3 *k*-fold Cross-validation

By default, the MNIST data set is divided into a training set and a smaller validation set. During every training epoch, the model is fed the training data and allowed to update its weights, learning to fit the training data more accurately as it goes. However, validating the model's performance on an independent data set—one not used for training—is critical, and is performed at the end of every epoch to ensure that the model can generalize to new data. (This validation set is separate from the test data set presented to the model after development is complete.)

Having separate training and validation sets keeps things simple. However, there are many more sophisticated alternatives,[11] including *k*-fold cross-validation, which is the method we use.

In *k*-fold cross-validation, the training and validation data are not kept separate from the beginning. Instead, the combined data is separated into *k* equal subsets, or "folds." Rather than training and validating only one model, we then train *k* separate models. For each such model, one of the *k* folds is reserved as the validation data set, and the others are combined into one training set. The final validation loss of the model, after training, is the average of the validation losses of all *k* separately trained and validated models. Not only does this give a better estimation of the model's true performance, the standard deviation of this average validation loss provides a measure of the model's stability.

When the *k*-fold cross-validation is complete, if performance is satisfactory, the model is retrained on all the data (a less computationally expensive task) to give the final model.

2.3.4 Hyperparameter Optimization

We chose the model hyperparameters (another name for the model architecture) for our MNIST classifier somewhat arbitrarily. If we had specified a different number of layers or kernels per layer, or different kernel or pool sizes, the classifier would likely have performed well anyway. However, some sets of hyperparameters are better than others, depending on the details of a particular problem. In order to find good hyperparameters, we included an optimization function using the

Approved for public release; distribution is unlimited.

`gp_minimize` function from Python's Scikit-Optimize library, which allows computationally costly functions to be minimized efficiently.

Rather than being set from the beginning, some hyperparameters (dropout rate, d; the number of kernels per layer, n_i; and the pool size per layer, p_i) are given an allowed range, while maintaining the basic architecture of the original MNIST classifier, as shown in Table 1. To decrease the number of hyperparameters to optimize over, we set the kernel size of each convolution layer equal to 1 plus twice the pool size of the corresponding max pooling layer. This ensures that the data is downsampled on a scale smaller than the size of the convolutional filters used in the previous layer.

Table 1 Hyperparameter ranges

Hyperparameter	[min, max]
dropout rate, d	[0, 0.25]
# kernels in conv. layer i, n_i	[8, 64]
pool size after conv. layer i, p_i	[2, 16]

Additionally, downsampling limits the sizes of the pools in the model, such that their product must be less than the original number of data points (8,000). If the chosen pool sizes violate this rule, the deepest convolution/max pooling layer pair is removed from the model, repeating until the rule is satisfied. We allow the number of convolution/max pooling layer pairs to vary between 3 and 5 as a result.

The `gp_minimize` function chooses specific hyperparameter values within the allowed ranges and fully trains a k-fold cross-validated model for each of them. The hyperparameters of the first few models are chosen randomly. The `gp_minimize` function chooses subsequent hyperparameter sets by approximating the loss function of the CNN as a Gaussian process, allowing it to estimate the change in hyperparameters most likely to decrease the loss function. At the end of the optimization process, the models that had the lowest average validation loss are trained on all the data to give the final model. When selecting the best model, we check that the standard deviation of the validation losses obtained across each fold for that model is neither unusually large compared to the standard deviation of other tested models, nor larger than the average.

Approved for public release; distribution is unlimited.

3. Results and Discussion

After optimization, we find that the best model, subject to these constraints, has the hyperparameter values displayed in Table 2.

Table 2 Optimized hyperparameter values

Hyperparameter	Value
d	0.04
n_1	8
n_2	35
n_3	58
n_4	12
n_5	2
p_1	7
p_2	8
p_3	3
p_4	13
p_5	2

This network, trained on $k = 14$ folds (one for each data set), gives a validation loss (mean squared error between predicted and measured t/t_c) of 0.6 ± 0.5, indicating that the model was not able to generalize very well from the training data. This is confirmed by Fig. 4, where we plot the predicted t/t_c versus the t/t_c obtained from the measurement timestamps. There are 14 plots, one for each cross-validation fold.

Approved for public release; distribution is unlimited.

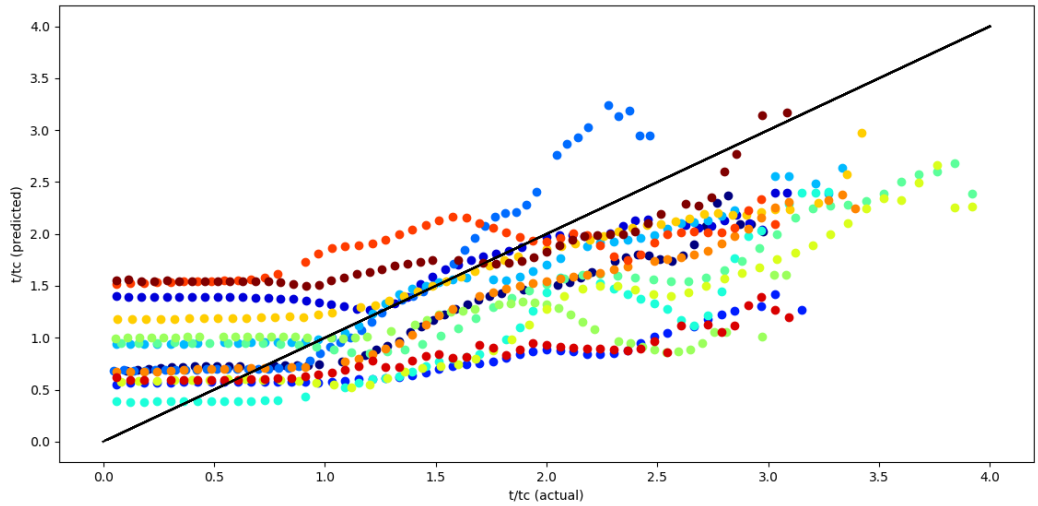

Fig. 4 Predicted vs. actual t/t_c for the 14 cross-validation data sets. The black line has a slope of 1.

As shown in Fig. 4, however, the model does pick out the most important features of the data. To begin with, every measurement begins with a flat plateau for $0 < t/t_c < 1$, indicating that the model cannot differentiate between acoustic signals in the absence of a crack. However, at about $t/t_c = 1$, the predicted value begins to change.

For $t/t_c < 1$, the predicted t/t_c is not consistent between experiments, with average and standard deviation 0.9 and 0.4, respectively. Further, the line of best fit to the points with $t/t_c > 1$ has a slope of only 0.7. (If the model performed perfectly, every point in Fig. 4 would lie on a line with slope 1 and intercept 0.) Moreover, the predicted values for several data sets oscillate rather than increase monotonically. Accurately predicting the "time since a crack has formed" is not possible in this study.

The same points hold true for the test data. We held back two data sets (i.e., true test data) separate from the training/validation data upon which we optimized and trained. After optimizing and cross-validating, we retrained the optimized network on the entire cross-validation set, then tested that model on the held-back data sets. Figure 5 shows the predicted t/t_c versus the t/t_c obtained from the measurement timestamps.

Both test sets are flat for an initial period before a crack forms then increase, although only one does so monotonically, and neither follows the line with slope 1.

Approved for public release; distribution is unlimited.

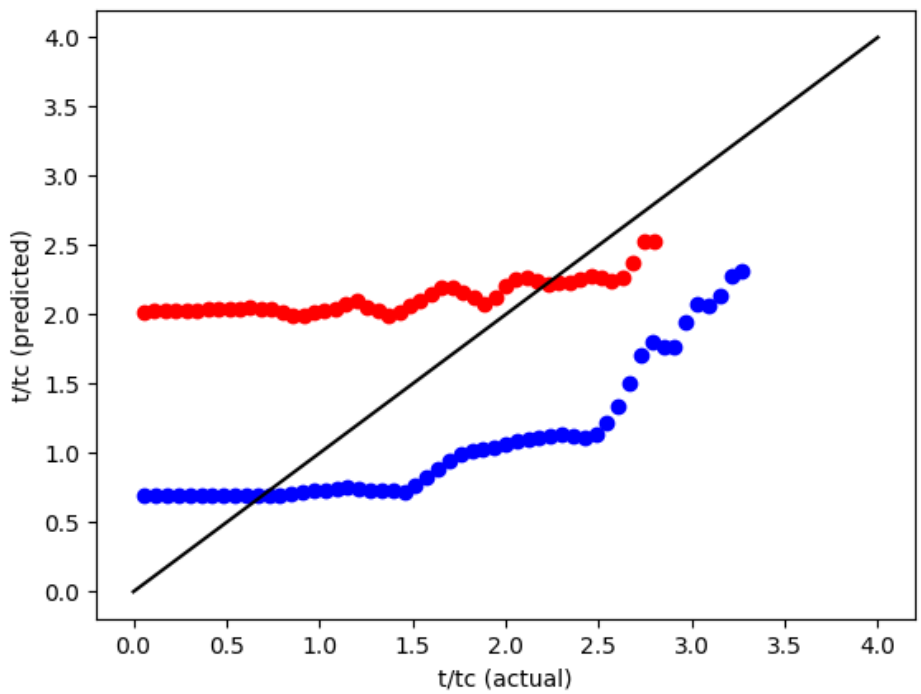

Fig. 5 Predicted vs. actual t/t_c for the two test data sets. The black line has a slope of 1.

However, if the only desired property of the model is to determine whether a crack has formed or not, it actually performs fairly well. The predicted value of t/t_c at $t/t_c = 0$, whatever that predicted value is, remains constant until a crack forms. At this point it changes, usually increasing by a factor of about 2 (varying from experiment to experiment) as the crack grows. One baseline measurement, taken before the crack forms, provides enough information in most cases to determine with reasonable confidence whether a crack has formed at some future time, even without any prior knowledge as to the value of t_c for that experiment. The baseline can even be taken after the fatiguing process is mostly complete. Perhaps this problem would be better recast as a classifier, with the two possible states "cracked" and "not cracked."

4. Conclusions

We have broken down the surprising similarity between a simple image classifier and a signal regressor in a way that emphasizes the universality of the problems to which a CNN can be applied. Specifically, though, the input shapes are, on the surface, very different (2-D, 1-channel images vs. 1-D, 72-channel data sets), both are ultimately just arrays. The CNN does not care about the shape of the array it receives.

Approved for public release; distribution is unlimited.

Despite our very simple approach to the problem, we obtained a model that, in qualitative terms, performed surprisingly well. Given acoustic data from a baseline measurement performed on a metal plate during fatigue experiments, the model can determine whether a crack has formed, although it cannot reliably quantify the extent of the crack or the remaining time to complete failure of the plate. Nevertheless, we regard the outcome as at least a qualified success.

Perhaps more importantly, we demonstrate that a CNN can be developed to approach problems vastly different from image recognition, which is by far the most common application. Said another way, a CNN cannot tell whether a given data set represents an image or not, and there is no reason not to use CNNs to model phenomena besides images.

Finally, in the Appendix, we have provided an example CNN implementation in Python using several powerful tools developed over the years by the deep learning community, but not often collected together in one place. We hope that fellow users of machine learning will find these helpful as they look for ways to apply CNNs to their own work.

Approved for public release; distribution is unlimited.

5. References

1. Goodfellow I, Benjio Y, Courville A. Deep Learning. Cambridge (MA): MIT Press; 2016 [accessed 2018 Mar 15]. https://www.deeplearningbook.org.

2. LeCun Y, Yoshua B. Convolutional networks for images, speech, and time-series. In: Arbib MA, editor. The handbook of brain theory and neural networks. Cambridge (MA): MIT Press; 1995. p. 255–258.

3. Malek S, Melgani F, Bazi Y. One-dimensional convolutional neural networks for spectroscopic signal regression. J. Chemom. 2018;32(5):e2977.

4. O'Shea T, Roy T, Clancy TC. Over the air deep learning based radio signal classification. IEEE J. Sel. Topics Signal Process. 2018 Feb;12(1):168−179.

5. van den Oord et al. WaveNet: a generative model for raw audio. arXiv:1609.03499.

6. Iglesias EE, Haynes RA, Shiao C. Inspection correlation study of ultrasonic-based in situ structural health monitoring monthly report for December 2014–January 2015. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2015 May. Report No.: ARL-TN-0671.

7. Lecun Y, Bottou L, Bengio Y, Haffner P. Gradient-based learning applied to document recognition. Proc. IEEE. 1998 Nov;86(11):2278–2324.

8. Bishop CM. Pattern recognition and machine learning. New York (NY): Springer; 2006. (Information Science and Statistics).

9. Raghu M, Poole B, Kleinberg J, Ganguli S, Sohl-Dickstein J. On the expressive power of deep neural networks. In: Precup D and Teh YW, editors. ICML 2017. Proceedings of the 34th International Conference on Machine Learning; 2017 Aug 6–11; Sydney, NSW, Australia. Brookline (MA): Proceedings of Machine Learning Research; c2017. p. 2847–2854.

10. Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R. Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. 2014;15:1929–1958.

11. Arlot S, Celisse A. A survey of cross-validation procedures for model selection. Stat. Surv. 2010;4:40–79.

Approved for public release; distribution is unlimited.

Appendix. Python Scripts for a 72-Channel Acoustic Signal Regressor

Approved for public release; distribution is unlimited.

The signal regressor shown below is written in Python and uses the Keras machine learning library. More information about Keras, including documentation and required libraries, can be found at https://keras.io. We use TensorFlow as the backend in the following code.

The code incorporates the Scikit-Optimize (skopt) module. More information about skopt, including documentation and required libraries, can be found at https://scikit-optimize.github.io.

Additionally, for clarity and ease of use, it is broken into three parts: (1) a top-level script, `CNN_acoustic.py`; (2) a wrapper for the model, `CNN_model.py`; and (3) an empty file, `CNN_globals.py`, used to shuttle global variables back and forth between the first two.

(1) `CNN_acoustic.py`

```
"""MODULES"""

import numpy as np
import random as rn
import os
import sys
import keras
import pickle
from skopt import gp_minimize

# If you are not storing all three scripts in the same directory, change the path below to include the location of CNN_model.py and CNN_globals.py.
sys.path.append('.')
import CNN_model
import CNN_globals

#######################################################

"""PARAMETERS"""

# if optimize=True, run skopt to optimize the hyperparameters. (Do this first.)
# if optimize=False, train the model on specified hyperparameters. (Do this second.)
optimize = True
CNN_globals.optimize = optimize

```

```
# if optimize=False, this is where you input the hyperparameters on which to train your final model.
if not optimize:
  # The dropout rate.
  d = 0

  # The number of filters in each of the 5 layers.
  nfilter1 = 8
  nfilter2 = 8
  nfilter3 = 8
  nfilter4 = 8
  nfilter5 = 8

  # The pool size in each pooling layers.
  psize1 = 2
  psize2 = 2
  psize3 = 2
  psize4 = 2
  psize5 = 2

  # All the parameters go into a list.
  x = [d,
    nfilter1, nfilter2, nfilter3, nfilter4, nfilter5,
    psize1, psize2, psize3, psize4, psize5]

# if optimize=False, the model must be saved after training and validation on each fold.
if not optimize:
  save_dir = 'saved_acoustic_models'
  model_name = 'acoustic_model_k%s.h5'
  model_path = os.path.join(save_dir, model_name)

  CNN_globals.save_dir = save_dir
  CNN_globals.model_name = model_name
  CNN_globals.model_path = model_path

# You can use previous optimizations for the same problem to inform the current iteration. This means you can run multiple parallel optimizations and use this feature to combine them all afterwards.
# if import_prev_results=True, you are using the results of previous optimizations to inform the optimizer.
```

```
# if import_prev_results=False, you are not.
import_prev_result = False

if import_prev_results:
  # The paths to the res objects you got from the previous optimizations go in the following list. (These are pickled.)
  prev_optims = []

# How many folds are you going to cross-validate over?
num_splits = 2
CNN_globals.num_splits = num_splits

# How many epochs are you going to train over?
num_epochs = 40
CNN_globals.num_epochs = num_epochs

# Early stopping keeps the model from running for longer than it needs to and helps avoid overfitting.
early_stop = keras.callbacks.EarlyStopping(
  monitor='val_loss',
  min_delta=0,
  patience=20,
  verbose=0,
  mode='auto')
callbacks_list = [early_stop]
CNN_globals.callbacks_list = callbacks_list

# experiments.pkl is a pickled list of paths, each one pointed at the directory containing the data from one experiment. Each directory contains 2 numpy files, one each for the data and labels, named X.npy and Y.npy, respectively.
experiments = pickle.load(open(
  '/some_directory/experiments.pkl', rb'))

# Number of random starts during the optimization.
if optimize:
  num_rands = 1

# Number of times to run the optimizer (including the random starts).
if optimize:
```

```
  num_calls = 2
  CNN_globals.num_calls = num_calls

# Number of files per batch during training.
batch = 16
CNN_globals.batch = batch

# Each measurement is 8,000 points long and 72 channels deep.
img_length = 8000
img_depth = 72

CNN_globals.img_length = img_length
CNN_globals.img_depth = img_depth

# The outputs of the optimizer will be stored in the following dictionary:
outputs = {
  'hyperparameters':[],
  'train_loss_mean':[],
  'train_loss_std':[],
  'val_loss_mean':[],
  'val_loss_std':[]}
CNN_globals.outputs = outputs

#######################################################

"""IMPORT PREVIOUS RESULTS"""
if import_prev_results:

  # The previous results will go in these empty lists:
  x_0 = []
  y_0 = []

  # Locate and unpickle the res objects from the previous optimizations:
  for prev in prev_optims:
    opt_hist = pickle.load(open(prev, 'rb'))

    # Load the hyperparameters and validation loss values into x_0 and y_0, respectively.
    for i in range(len(opt_hist['hyperparameters'])):
      temp_list =[
        [opt_hist['hyperparameters'][i][0]],
```

```
        opt_hist['hyperparameters'][i][1],
        opt_hist['hyperparameters'][i][2]]
      temp_list = [
        val for sublist in temp_list for val in
          sublist]
      x_0.append(temp_list)

    y_0.append(opt_hist['val_loss_mean'])

else:
  x_0 = None
  y_0 = None

#######################################################

"""LOAD AND PREPROCESS EXPERIMENTAL DATA"""

# Empty lists to accept the data.
X_loaded = []
Y_loaded = []

# Load the data. All of the experiment names in
experiments.pkl correspond to one directory that
contains the data. The data is separated into 2 files,
X_ExperimentNumber.npy and Y_ExperimentNumber.npy.
# We unfortunately have not been given permission to
release the actual data.
for experiment in experiments:
  X_loaded.append(np.load(experiment + 'X.npy'))
  Y_loaded.append(np.load(experiment + 'Y.npy'))

# Some experiments lasted slightly less time than
others. Because the data all has to be in array format
(due to the way KFold works) we have to truncate the
longer experiments. Basically, data points are removed
randomly from the longer experiments until all have
the same length as the shortest experiment.
# An alternative would be to copy random points from
the shorter experiments, until all are the same length
as the longest experiment.

# What were the original lengths of each experiment?
original_lengths = []

```

```
for i in range(len(experiments)):
  original_lengths.append(len(Y_loaded[i]))

# What is the length of the shortest experiment?
min_length = min(original_lengths)
CNN_globals.min_length = min_length

# Randomly delete measurements from the longer experiments until all have the same length.
for i in range(len(experiments)):
  while len(X_loaded[i]) > min_length:
  rand_temp = rn.randint(0, len(X_loaded[i]) - 1)
  X_loaded[i] = np.delete(X_loaded[i], rand_temp, 0)
  Y_loaded[i] = np.delete(Y_loaded[i], rand_temp, 0)

# How many measurements were lost from each experiment when X and Y were converted to arrays?
measurements_lost = []
for experiment in experiments:
  measurements_lost.append(original_lengths[
    experiments.index(experiment)] - min_length)

# Now that they're all the same length, convert them into arrays.
X_loaded = np.asarray(X_loaded)
Y_loaded = np.asarray(Y_loaded)

# Shuffle the experiments.
shuffler = np.arange(len(X_loaded))
np.random.shuffle(shuffler)

# These are the final X and Y that will be used to train and cross-validate the model.
X = X_loaded[shuffler]
Y = Y_loaded[shuffler]

CNN_globals.X = X
CNN_globals.Y = Y

#######################################################

"""RUN THE MODEL"""

# if optimize=True, use gp_minimize to find the
```

```
optimal hyperparameters.
# if optimize=False, train the model using the
optimized hyperparameters.

if optimize:

  # Set the upper and lower bounds for the
hyperparameters to be optimized.
  d_min = 0
  d_max = 0.25
  nfilter_min = 8
  nfilter_max = 64
  psize_min = 2
  psize_max = 16

  res = gp_minimize(CNN_model.runModel,
          [(d_min, d_max), # d
          (nfilter_min, nfilter_max), # nfilter1
          (nfilter_min, nfilter_max), # nfilter2
          (nfilter_min, nfilter_max), # nfilter3
          (nfilter_min, nfilter_max), # nfilter4
          (nfilter_min, nfilter_max), # nfilter5
          (psize_min, psize_max), # psize1
          (psize_min, psize_max), # psize2
          (psize_min, psize_max), # psize3
          (psize_min, psize_max), # psize4
          (psize_min, psize_max), # psize5
          ],
          n_calls=num_calls,
          n_restarts_optimizer=1,
          verbose=True,
          x0=x_0,
          y0=y_0,
          n_random_starts=num_rands)

  print(res.x)
  print(res.fun)

  with open('res.pkl', 'wb') as f:
    pickle.dump(res, f)

else:
  CNN_model.runModel(x)

```

```
for experiment in experiments:
  print('%s measurements skipped from %s (
    original length: %s measurements).' % (
    measurements_lost[experiments.index(experiment)],
    experiment,
    original_lengths[experiments.index(experiment)]))
```

Approved for public release; distribution is unlimited.

(2) CNN_model.py

```
"""MODULES"""

import keras
from sklearn.model_selection import KFold
import os
import numpy as np
import sys
import pickle

# If you are not storing all three files in the same directory, change the path below to include the location of CNN_model.py and CNN_globals.py.
sys.path.append('.')
import CNN_globals

#######################################################

"""MODEL"""

def runModel(x):

  # The dropout rate.
  d = x[0]

  # The number of filters in each of the 5 layers.
  nfilter1 = int(x[1])
  nfilter2 = int(x[2])
  nfilter3 = int(x[3])
  nfilter4 = int(x[4])
  nfilter5 = int(x[5])

  # The pool size in each pooling layer.
  psize1 = int(x[6])
  psize2 = int(x[7])
  psize3 = int(x[8])
  psize4 = int(x[9])
  psize5 = int(x[10])

  # We do not keep kernel and pool sizes independent.
  ksize1 = 2 * psize1 + 1
  ksize2 = 2 * psize2 + 1
  ksize3 = 2 * psize3 + 1
```

Approved for public release; distribution is unlimited.

```
  ksize4 = 2 * psize4 + 1
  ksize5 = 2 * psize5 + 1

  # Display only the relevant parameter values. Remember, the data cannot be downsampled to less than 1 point.
  if psize1 * psize2 * psize3 * psize4 > 8000:
    print('dropout rate = ', d)
    print('# filters = ',
      nfilter1, nfilter2, nfilter3)
    print('kernel sizes = ', ksize1, ksize2, ksize3)
    print('pool sizes = ', psize1, psize2, psize3)
    print('-----------------------------------------')

  elif psize1 * psize2 * psize3 * psize4 < 8000 and psize1 * psize2 * psize3 * psize4 * psize5 > 8000:
    print('dropout rate = ', d)
    print('# filters = ',
      nfilter1, nfilter2, nfilter3, nfilter4)
    print('kernel sizes = ',
      ksize1, ksize2, ksize3, ksize4)
    print('pool sizes = ',
      psize1, psize2, psize3, psize4)
    print('-----------------------------------------')

  elif psize1 * psize2 * psize3 * psize4 * psize5 < 8000:
    print('dropout rate = ', d)
    print('# filters = ',
      nfilter1, nfilter2, nfilter3, nfilter4,
      nfilter5)
    print('kernel sizes = ',
      ksize1, ksize2, ksize3, ksize4, ksize5)
    print('pool sizes = ',
      psize1, psize2, psize3, psize4, psize5)
    print('-----------------------------------------')

  # Define the model.
  model_input = keras.layers.Input(
    shape=(CNN_globals.img_length,
    CNN_globals.img_depth))

  conv_1 = keras.layers.Conv1D(
```

```
      nfilter1,
      kernel_size=ksize1,
      strides=1,
      activation='relu',
      padding='same',
      name='conv_1')(model_input)
    pool_1 = keras.layers.MaxPool1D(
      pool_size=psize1,
      name='pool_1')(conv_1)
    dropout_1 = keras.layers.Dropout(
      d,
      name='dropout_1')(pool_1)

    conv_2 = keras.layers.Conv1D(
      nfilter2,
      kernel_size=ksize2,
      strides=1,
      activation='relu',
      padding='same',
      name='conv_2')(dropout_1)
    pool_2 = keras.layers.MaxPool1D(
      pool_size=psize2,
      name='pool_2')(conv_2)
    dropout_2 = keras.layers.Dropout(
      d,
      name='dropout_2')(pool_2)

    conv_3 = keras.layers.Conv1D(
      nfilter3,
      kernel_size=ksize3,
      strides=1,
      activation='relu',
      padding='same',
      name='conv_3')(dropout_2)
    pool_3 = keras.layers.MaxPool1D(
      pool_size=psize3,
      name='pool_3')(conv_3)
    dropout_3 = keras.layers.Dropout(
      d,
      name='dropout_3')(pool_3)
```

```
  # If the pool sizes in the early layers are large enough, the model may be truncated after 3 or 4 layers.

  if psize1 * psize2 * psize3 * psize4 < 8000:
    conv_4 = keras.layers.Conv1D(
      nfilter4,
      kernel_size=ksize4,
      strides=1,
      activation='relu',
      padding='same',
      name='conv_4')(dropout_3)
    pool_4 = keras.layers.MaxPool1D(
      pool_size=psize4,
      name='pool_4')(conv_4)
    dropout_4 = keras.layers.Dropout(
      d,
      name='dropout_4')(pool_4)

    if psize1 * psize2 * psize3 * psize4 * psize5 < 8000:
      conv_5 = keras.layers.Conv1D(
        nfilter5,
        kernel_size=ksize5,
        strides=1,
        activation='relu',
        padding='same',
        name='conv_4')(dropout_4)
      pool_5 = keras.layers.MaxPool1D(
        pool_size=psize5,
        name='pool_5')(conv_5)
      dropout_5 = keras.layers.Dropout(
        d,
        name='dropout_5')(pool_5)

  pool_global = keras.layers.GlobalAveragePooling1D(
    name='pool_global')(dropout_5)

  output = keras.layers.Dense(
```

```
    1,
    name='output')(pool_global)

  model.compile(
    loss='mean_squared_error',
    optimizer='adam')

  # Save the (random) initial weights to reset the model at the beginning of every fold.
  initial_weights = model.get_weights()

  print(model.summary())

  ####################################################

  """K-FOLD CROSS-VALIDATION"""

  # Each of the k folds divides the data into training and validation sets for that fold.
  Kfold = KFold(
    n_splits=CNN_globals.num_splits,
    shuffle=False)

  # Make a list to hold the training and validation losses for each fold.
  kfolds_results = []

  # These will be split into two lists.
  training_loss = []
  validation_loss = []

  # What is the number of the current fold?
  fold_num = 0

  ####################################################

  """TRAIN AND EVALUATE THE MODEL"""

  # Each channel in the training and validation sets are normalized by their respective standard deviations. This is done again for each fold.

  if not CNN_globals.optimize:
```

```
    Y_actual = {}
    Y_predict = {}

  for train_set, val_set in kfold.split(CNN_globals.X, CNN_globals.Y):

    # Increment the fold index.
    fold_num += 1
    print('Evaluating fold %s of %s.' % (
      fold_num, CNN_globals.num_splits))

    # Make an empty dictionary to receive the output later.
    output = {}

    # Separate out the training and validation sets for this fold.
    X_train = CNN_globals.X[train_set]
    X_val = CNN_globals.X[val_set]

    Y_train = CNN_globals.Y[train_set]
    Y_val = CNN_globals.Y[val_set]

    X_train_norm = np.zeros_like(X_train)
    X_val_norm = np.zeros_like(X_val)

    # Divide each channel by the standard deviation of all values in that channel. Do this separately for training and validation sets in each fold.
    for i in range(0,CNN_globals.img_depth):
      X_train_temp = X_train[:,:,:,i]
      X_val_temp = X_val[:,:,:,i]

      X_train_norm[:,:,:,i] = X_train_temp /
        np.std(X_train_temp)
      X_val_norm[:,:,:,i] = X_val_temp /
        np.std(X_val_temp)

    X_train_norm = X_train_norm.reshape(
      len(train_set) * CNN_globals.min_length,
      CNN_globals.img_length,
      CNN_globals.img_depth)
    X_val_norm = X_val_norm.reshape(
      len(val_set) * CNN_globals.min_length,
```

```
        CNN_globals.img_length,
        CNN_globals.img_depth)

    Y_train = Y_train.reshape(
        len(train_set * CNN_globals.min_length)
    Y_val = Y_val.reshape(
        len(val_set * CNN_globals.min_length)

    # Reset the initial weights at each fold.
    model.set_weights(initial_weights)

    # Each fold's output comes from training and validation of that fold's model.
    output = model.fit(
        X_train_norm,
        Y_train, batch_size=CNN_globals.batch,
        epochs=CNN_globals.num_epochs,
        verbose=1,
        shuffle=True,
        validation_data=(X_val_norm, Y_val),
        callbacks=CNN_globals.model_callbacks).history

    # Add each model's scores to its output.
    output['train_scores'] = model.evaluate(
        X_train_norm,
        Y_train,
        verbose=0)
    output['val_scores'] = model.evaluate(
        X_val_norm,
        Y_val,
        verbose=0)

    # For quantifying the model's performance, save the predicted and actual output values.
    if not CNN_globals.optimize:
        Y_predict[fold_num] = model.predict(X_val_norm)
        Y_actual[fold_num] = Y_val

    kfolds_results.append(output)

    ##################################################

    """SAVE MODEL AND WEIGHTS"""

```

```
    # Save each fold's model.
    if not CNN_globals.optimize:
      model.save(CNN_globals.model_path % fold_num)
      print('Saved trained model at %s.' % (
        CNN_globals.model_path))

  ##################################################

  """SCORE THE MODEL"""

  # Collect the training and validation loss from each fold.
  for i in range(CNN_globals.num_splits):
    training_loss.append(
      kfolds_results[i]['train_scores'])
    validation_loss.append(
      kfolds_results[i]['val_scores'])

  # Find their mean and standard deviation.
  training_loss_mean = np.mean(training_loss)
  training_loss_std = np.std(training_loss)

  validation_loss_mean = np.mean(validation_loss)
  validation_loss_std = np.std(validation_loss)

  # Print the final results of the cross-validation.
  print('training loss = %s' % training_loss)
  print('validation loss = %s' % validation_loss)
  print('best training loss = %s +/- %s' % (
    training_loss_mean, training_loss_std))
  print('number of folds = %s' % (
    CNN_globals.num_splits))

  ##################################################

  """SAVE OPTIMIZATION HISTORY"""

  # Save the hyperparameters from each iteration of the optimizer.
  if CNN_globals.optimize:
    CNN_globals.outputs['hyperparameters'].append(
      [[d],
      [nfilter1, nfilter2, nfilter3, nfilter4,
      nfilter5],
```

```
        [ksize1, ksize2, ksize3, ksize4, ksize5],
        [psize1, psize2, psize3, psize4, psize5]])
      CNN_globals.outputs['train_loss_mean'].append(
        training_loss_mean)
      CNN_globals.outputs['train_loss_std'].append(
        training_loss_std)
      CNN_globals.outputs['val_loss_mean'].append(
        validation_loss_mean)
      CNN_globals.outputs['val_loss_std'].append(
        validation_loss_std)

      # Save the optimization history every 5 iterations or when it has run through CNN_globals.num_calls iterations.
      if len(CNN_globals.outputs['val_loss_mean']) % 5 is 0 or len(CNN_globals.outputs['val_los_mean']) is CNN_globals.num_calls:
        with open('optimization_history.pkl', 'wb') as f:
          pickle.dump(CNN_globals.outputs, f)

    ##################################################

    """RETURN THE PARAMETER TO BE OPTIMIZED"""

    if CNN_globals.optimize:
      return(validation_loss_mean)
```

(3) CNN_globals.py

```
# This file is empty. Global variables are written
into it and read out by CNN_acoustic.py and
CNN_model.py.
```

List of Symbols, Abbreviations, and Acronyms

1-D	1-dimensional
2-D	2-dimensional
ARL	US Army Research Laboratory
CNN	convolutional neural network
MNIST	Modified National Institute of Standards and Technology
RGB	red, green, and blue

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIR ARL IMAL HRA RECORDS MGMT RDRL DCL TECH LIB
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
9 (PDF)	RDRL CIH S D SHIRES RDRL CIH C E CHIN M LEE RDRL SLE W M MARKOWSKI A BEVEC RDRL VTM M COATNEY A HALL R HAYNES RDRL VTP A HOOD

Approved for public release; distribution is unlimited.