Artificial Neural Networks for Jet Acoustic Prediction

Brian Whiteaker[1]
NASA Glenn Research Center, Cleveland, OH, 44135

Experiments were carried out by NASA Glenn Research Center's Aero-Acoustic Propulsion Laboratory to obtain acoustic data arising from jet surface interactions. Seven different configurations were tested and the acoustic signal above 1/12 octave was recorded at 24 locations in an arc above the jet source. This data was used to explore the application of machine learning techniques to predict sound levels. Isoforests anomaly detection was used in data exploration to examine outliers, and various artificial neural network (ANN) architectures from both previous and new work. Robust prediction results were observed from the networks considered. Feature selection and anti-overfitting techniques were employed in model design. Recent work in the area of uncertainty quantification of ANN outputs was employed. This effort culminated in the development of a command line tool which accesses a deep neural network trained on the selected configuration data, and provides diagnostic information along with prediction uncertainty quantification.

Nomenclature

Angle = angle of observation for the microphones
ANN = artificial neural network
DNN = deep neural network (hidden layers 64/128/128/64/87)
dB = decibels
FF = feedforward neural network (hidden layer 64)
hE = radial distance from jet center line to surface
Ma = acoustic Mach number (ideally expanded jet velocity/speed of sound at ambient conditions)
Mj = local jet Mach number (jet velocity/speed of sound at local jet conditions)
MSE = mean squared error
NPR = nozzle pressure ratio (jet total pressure/ambient pressure)
TTR = jet total temperature ratio
TSR = jet static temperature ratio
xE = axial distance from jet exit to surface trailing edge (inches)
xJ = jet potential length

I. Introduction

The turbulent flow of jet engine exhaust generates high sound levels in the vicinity of any jet aircraft. Research into ways of reducing the impact of jet acoustic noise levels on surrounding environments is an ongoing process. As an example, a useful possibility arising from this knowledge could be designing some form of noise cancellation into the aircraft itself to help reduce noise. Sound levels are studied in air traffic corridors of major cities to determine effects on the public. Likewise the effects on the public as passengers within the aircraft are also considered. To achieve a better understanding of jet acoustics, different perspectives can be useful. In recent years machine learning tools are becoming commonplace and continually increase their effectiveness to optimize and extend use of a "Big Data" approach. With the dataset accumulated from the Aero Acoustic Laboratory we have the opportunity to explore this approach as applied to acoustic noise predicition. In this exploration, tools from the field of machine learning (ML) were used to make predictions of jet surface acoustic noise interactions.

[1] Intern, VE, NASA Glenn Research Center, and University of California at San Diego.

For machine learning tools this paper explores the use of artificial neural networks (ANN) in the regression setting. This exploration supports development of an ANN based command line tool for predicting noise levels, given some configuration, at a particular frequency. The tables provided in the following sections show a progression from a simple network to a more complex network and the MSE loss performance of various choices. The values in these tables were collected with a fixed number of epochs (full passes through the training set), and were averaged over 5 repeated runs having random initializations of the network and random selections of the training and test data. Each of tables 1-6 progress from learning rate of 0.1 down to 0.001 to evaluate the performance of the models using fixed learning rates. Tables 7 and 8 give performance using the gradient descent optimizer Adam having weight decay enabled. The FF network is a smaller hidden layer network (h=64) which is used as a performance reference to the more complex network. A more complex network, called DNN (h=64/128/128/46), is compared because of its greater capacity to learn from more complex data set. DNN is implemented in the command line tool for this reason. These models were ran using *Pytorch* with the Nvidia P-100 graphical processing units on the cluster belonging to NASA Glenn Research Center Graphics and Visualization Lab. Use of these high performance computing resources is valuable to the time consuming model prototyping process.

A. Experiment

Jet surface interactions arise when various surfaces on the aircraft cause reflection, shielding or other effects. In the experiments performed at NASA Glenn Research Center both round and rectangular nozzle shapes were used. The rectangular nozzles had aspect ratios: 2:1, 4:1, 8:1. Including the tests with round nozzles, a surface was introduced in positions of shielding such that it lies between the microphones and jet and also in a reflecting position on the far side of the jet from the microphones. In example, this can simulate aircraft designs having engines below the wing and passengers.

For the actual experimental tests, the microphones were position in an arc centered above the jet nozzle at angles every 5 degrees from 50 degrees to 165 degrees. The reflecting plane surface was also varied in position vertically, and measured with hE. The distance from nozzle to trailing edge of the surface was measured with xE. Beyond these nozzle type, shielding or reflecting, and geometric configurations, there were 87 frequencies recorded by the microphone. It is these frequencies that we target for prediction by the model.

B. Data

Some transformations were applied to the data to create dimensionless ratios for ease of scaling to any new designs resulting from this work. The transformations performed in the working code are:

$$hE = \frac{hE}{xE}$$

to eliminate inches,

$$dE = \sqrt{4 * NozzleArea/\pi}$$

to standardize nozzle opening area,

$$xE = xE/xJ$$

to measure by length of jet potential core.

In the ANN experiments, the data was subset into configuration sets. These configurations are engine nozzle aspect ratio by shield or reflect$\{1, 2, 4, 8\} \times \{0,1\}$. So, a round nozzle with the surface in a reflecting position, may be referred to as 1-1 (nozzle shape-shield /reflect). Each subset was then broken into training and test sets with *SkLearn train_test_split*, with a split of 60%/40%. The data was rescaled using the Python library *SkLearn MinMaxScaler*. In the process of training, the data was fed to the network in mini-batches of size 50 observations per batch.

C. Exploration

In the course of exploration the variables which had correlation were removed due to being products of other variables in the set. Some variables showed large variances in relation to others. For example some were on the scale

of hundreds versus others which might be in a range from 0 to 2. There was structure evident such as multi-modality regardless of the range of variance and so min-max scaling was applied to all variables. Rescaling to the 0 to 1 range would still allow for structure to be retained unchanged at a consistent scale without inducing numerical error issues during modeling.

Use of the *SkLearn IsolationForest* anomaly detection algorithm indicated outliers both upstream and downstream of the nozzle. The goal in applying this algorithm was to identify any angles where the data might be difficult to model due to high numbers of outliers. There are greater counts of outliers at the angles upstream of the nozzle and downstream of the nozzle, near vertically above the nozzle the outlier counts are low. The upstream noise could be attributable to structures holding equipment used in the experiment. The *Sklearn IsolationForest* anomaly detection algorithm is useful for high dimensional outlier detection. It operates by partitioning each point in a space and creates a tree whose edges represent partitions required to place each point in its individual cube/leaf. The contamination parameter sets what percentage of the points the algorithm should expect to be anomalies. *Sklearn IsolationForest* was ran with contamination parameter set to 5%, 10%, and 20% with similar indications of outlier counts being higher upstream and downstream in a consistent fashion.

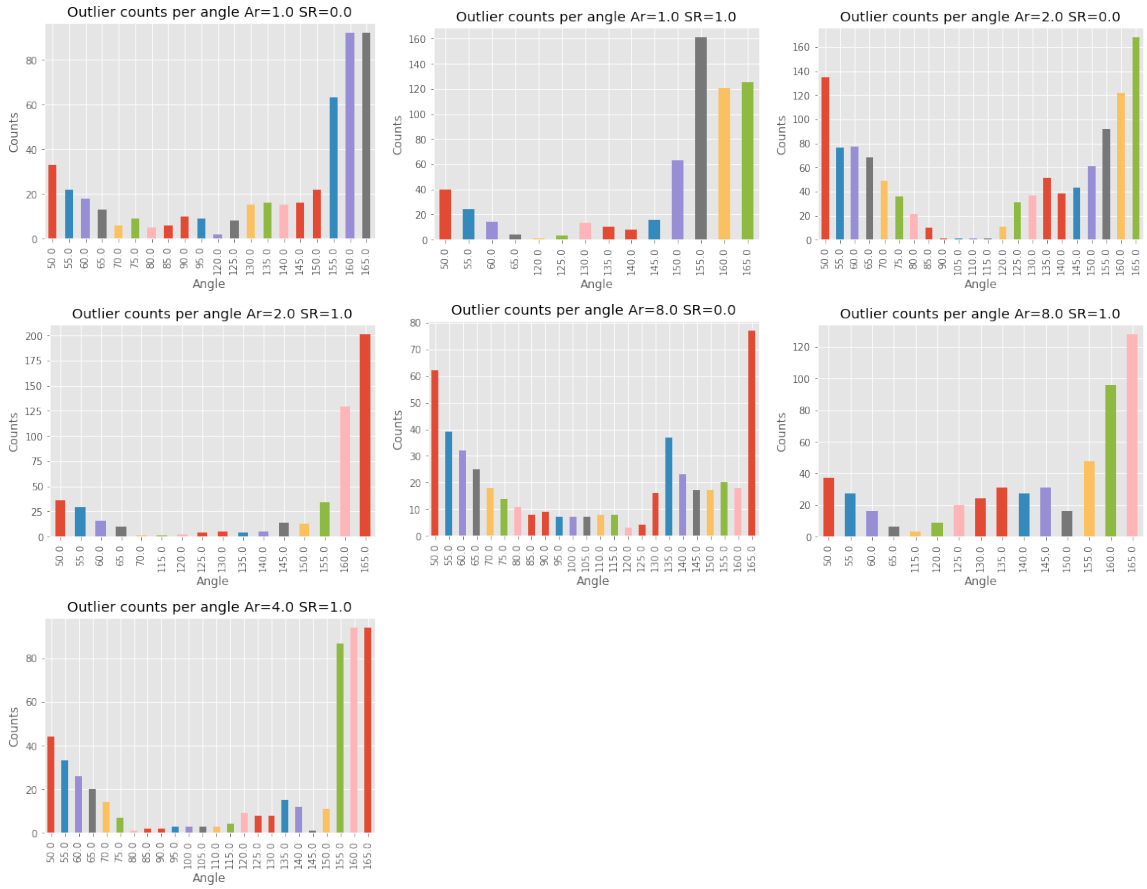


Figure 2. Anomaly detection. *These plots were generated using the IsolationForest algorithm. All seven configurations are shown with nozzle type given as aspect ratio(AR) and shield or reflecting(SR) where 1 is reflecting. The goal was to see where outliers challenging to model training might arise in the data. Some plots have less columns due to the value being zero, in which case they were excluded from plotting.*

D. Feature Engineering

Feature engineering is a hugely important aspect of data science that is commonly performed by experienced data scientists and allocated much effort. The skills to recognize what techniques to apply require development over time through work with data sets of different character. The refrain from these scientists is consistently that if the feature engineering is done well, even poor choice of learning algorithm will still perform quite well. Recognizing that this work would result in a tool for future learning on data from more complex configurations, it was decided to incorporate this expanded feature engineered set of variables. The thought is that use of anti-overfitting techniques

would strike a balance regarding an overly complex model which over-fit too easily, and a model whose complexity capacity can handle conceived future experiments with more complexity. The added benefit is that the data may also have nonlinear relationships which could be useful as basis functions in linear combinations formed by the network.

While ANN's are able to generate many transformations of variables which they may need, there are certain transformations which they cannot generate. Similar to a linear model, the network creates linear combinations of the input variables. They differ by passing the resulting value through the nonlinearity function(also called a "mashing" or "transfer" function) such as a ReLU or tanh before moving on to the next layer. To give the model more tools to work with it is useful to create/engineer nonlinear basis functions that it could not generate itself using the original basis functions (variables) in the dataset. In this work, nonlinear basis functions were added to the input space to expand the tools with which the ANN can use to model the data. Of the set $\{xE, hE, Ma, Mj, NPR, TTR, TSR, angle\}$ all were expanded to polynomials of second, third, and fourth degree with interaction terms included. The transformations sine, cosine, tangent, ln, and square root were also applied to each of these individually. Regarding the polynomials, an example of dataset column (variable) transformations would be $\{xE^2 hE^2, angle^4, \ln NPR,\ xE \cdot TTR, \ldots\}$. Use of *Pandas DataFrame* allows application of product and basic functions to entire variable columns, thereby creating new columns. This was extremely useful. The total count of input variables was 534 after dataset expanding transformations are applied. This addition of features resulted in an overall lowering of mean squared error on training and test sets by providing access to more degrees of freedom for finer grained modeling.

Table 1. MSE for FF h=64, learning_rate=0.1 @ 500 epochs, Asterisk is feature engineering.

Test Config	FF ReLU	FF ReLU*	FF Tanh	FF Tanh*
1-1	1.73	0.51	0.71	1.35
1-0	1.43	0.72	1.10	2.06
2-1	1.88	0.63	0.77	1.78
2-0	4.03	1.42	2.03	3.60
4-1	1.71	0.77	1.04	1.70
8-1	1.78	0.93	1.14	1.85
8-0	3.24	1.25	1.73	4.18

Table 2. MSE for DNN h=64/128/128/64, learning_rate=0.1 @ 500 epochs, Asterisk is feature engineering.

Test Config	DNN ReLU	DNN ReLU*	DNN Tanh	DNN Tanh*
1-1	90.25	90.84	0.87	1.47
1-0	90.87	40.13	1.41	3.19
2-1	85.37	53.66	1.04	1.32
2-0	103.69	48.90	5.86	9.12
4-1	85.76	76.74	1.45	1.58
8-1	87.06	26.67	1.63	2.09
8-0	99.84	49.71	3.73	4.66

E. Architecture

Various architectures were experimented on using the *Pytorch* deep learning framework. Two were inspired by earlier work from [5][7] and were the starting point for this work. The final network arrived at by [5] consisted of 3 hidden layers with node counts of 30/50/20. The work of [5] used a different dataset and the nozzle types were rectangular only. The second inspiration [2], led to a single hidden layer architecture with 64 nodes. This work was performed on the same dataset used in this paper, and so the output dimension of 87 corresponds to the 87 frequencies predicted. Most prototyping efforts were ran on the 64 node model due to time constraints for training. Other architectures used were multiple hidden layer models with node counts 32/32/32, 128, 512, 128/128/128, 64/32/32. The final architecture used in the command line tool is 64/128/128/64. This architecture easily over-fits the data in most cases. This was by design and is meant to be counteracted by techniques such as early stopping when a threshold MSE on the test set is achieved, dropout, k-folds cross validation. LASSO was passed over as a design decision due to use of uncertainty quantification. After extensive model testing it did not appear necessary other than for exploring feature selection by the network, which is a parallel effort to development of the command line tool.

Table 3. MSE for FF h=64, learning_rate=0.01 @ 500 epochs, Asterisk is feature engineering.

Test Config	FF ReLU	FF ReLU*	FF Tanh	FF Tanh*
1-1	0.30	0.64	0.40	0.84
1-0	0.33	1.17	0.56	1.17
2-1	0.43	0.86	0.55	0.98
2-0	1.15	1.45	1.27	2.12
4-1	0.51	1.12	0.75	1.03
8-1	0.61	1.51	0.91	1.33
8-0	0.88	2.31	1.40	2.10

Table 4. MSE for DNN h=64/128/128/64, learning_rate=0.01 @ 500 epochs, Asterisk is feature engineering.

Test Config	DNN ReLU	DNN ReLU*	DNN Tanh	DNN Tanh*
1-1	0.27	0.18	0.32	0.73
1-0	1.02	0.33	0.71	2.01
2-1	0.34	0.24	0.54	0.69
2-0	0.76	0.61	1.68	2.12
4-1	0.47	0.34	0.67	1.43
8-1	0.47	0.42	1.09	1.22
8-0	0.81	0.73	2.36	2.30

As for nonlinearity functions various *Pytorch* functions were tested. These were *ELU, ReLU, LeakyReLU, Tanh* and of these *Tanh* applied with *BatchNorm1d* could achieve the lowest MSE (~0.03 on some configs) given high epoch counts. At this point the model absolutely over-fitted and so the MSE achievable through *ReLU* was deemed more than sufficient for our purposes. It is important to note that in the actual experiments a $\pm 0.5\ dB$ irreducible error present from the acoustic sampling equipment itself. So any ANN model achieving MSE below this value is most likely in the realm of overfitting at that point. Another consideration is the 1-dimensional linear spline approximation from *ReLU* versus the near 0-dim approximation from *Tanh* works more smoothly with the uncertainty tools developed by [Gal].

In all models we use *Pytorch MSELoss* to calculate the mean squared error loss on each pass or "epoch" of the data. Another choice of loss function used was *Pytorch SmoothL1Loss* which is a form of the Huber loss function. This loss function is meant to combine the best qualities of MSE and $\mathcal{L}^1$ loss. Huber loss presented difficulties in implementation with the uncertainty quantification. The performance difference was negligible and so MSE was the choice selected for the models.

Table 5. MSE for FF h=64, learning_rate=0.001 @ 500 epochs, Asterisk is feature engineering.

Test Config	FF ReLU	FF ReLU*	FF Tanh	FF Tanh*
1-1	0.85	5.44	0.45	1.06
1-0	1.08	7.65	0.66	2.08
2-1	1.29	7.81	0.73	1.44
2-0	2.00	11.25	1.64	2.67
4-1	1.17	7.85	0.77	1.44
8-1	1.53	9.41	1.03	1.74
8-0	1.88	13.14	1.34	2.97

Table 6. MSE for DNN h=64/128/128/64, learning_rate=0.001 @ 500 epochs, Asterisk is feature engineering.

Test Config	DNN ReLU	DNN ReLU*	DNN Tanh	DNN Tanh*
1-1	0.30	0.39	0.35	0.88
1-0	0.39	0.61	0.72	1.13
2-1	0.45	0.47	0.53	0.77
2-0	0.67	0.93	0.88	1.60
4-1	0.58	0.68	0.66	0.89
8-1	0.67	0.74	0.79	1.05
8-0	0.79	1.11	1.20	1.90

Overfitting is a major problem with use of complex models. Complex models have enough available "capacity" to outright memorize the data in the network. An indication of overfitting is most obvious when looking at loss plots. This manifests as the curve for test error diverging upward from a still decreasing training error cure. This is interpreted as the model is now memorizing the training data noise very specifically and so the model is not generalizing well to the test data which it is not trained on. The bias-variance tradeoff aims for a certain amount of variance in the model after training so that it can make predictions on unseen data, and not be stuck regurgitating the training data points. An example may be seen in table 7 when looking at the values of the FF ReLU column at 500 epochs. Compare this to table 9 FF ReLU at 2000 epochs and an increase in error is observed. Since these values are test error we know that the model ran for longer has reached its limits and is now over-fitting.

For over-fitting issues the command line tool uses early stopping and *Pytorch Dropout* modules at each of the inner layers with a probability of dropout set at 0.1. When examining feature selection least absolute shrinkage and selection operator (LASSO) was used to find a sparse basis of features used by the network. Without LASSO, the network is free to use as many of the 534 variables available as it would like, allowing it to cheat and memorize the dataset in different regions of the 534 dimensional input space. LASSO penalizes the network into using the most "relevant" variables in the model. Dropout randomly disconnects weights or "connections" in the network on each pass. This requires the network to spread information around. Without using dropout, the network might randomly choose to use one variable for multiple jobs, this may mean an over-dependence on a single variable. Dropout might disconnect this over used variable and the resulting loss will be high. So the network will spread its reliance to other variables. Particularly, it may retain reliance on variables that truly are valuable. In a sense LASSO and dropout techniques are somewhat counter to each other. Using these tools together can massively increase training time. Considering an input space expanded to 534 variables and the matrix multiplications needed for multiple layers, training time needed to convergence must increase.

Table 7. MSE for h=64, weight-decay=1e-5 (Ridge) @ 500 epochs.

Config	FF ReLU	FF ReLU*	FF Tanh	FF Tanh*
1-1	0.49	1.54	0.69	1.05
1-0	0.71	2.50	0.64	1.92
2-1	0.64	1.68	0.63	1.28
2-0	1.31	3.09	1.37	2.66
4-1	1.01	1.67	0.86	1.52
8-1	1.10	2.43	1.12	1.79
8-0	1.93	6.67	1.34	3.09

Table 8. MSE for h=64/128/128/64, weight-decay=1e-5 (Ridge) @ 500 epochs.

Config	DNN ReLU	DNN ReLU*	DNN1 Tanh	DNN Tanh*
1-1	0.29	0.37	0.37	0.80
1-0	0.40	0.54	0.55	1.18
2-1	0.47	0.48	0.53	0.72
2-0	0.64	0.85	1.00	1.68
4-1	0.52	0.63	0.59	0.84
8-1	0.66	0.79	0.83	1.12
8-0	0.77	1.04	1.15	1.79

E. Uncertainty Quantification

For any engineering work it is desirable to quantify the propagation of error or uncertainty. For work such as that carried out by NASA there is an error budget that is adhered to. With the growing widespread use of machine learning in many fields it will become important to find useful ways to quantify the uncertainty in predictions coming from a model. Recent work by [6] illustrates that certain deep neural networks can be treated as a Gaussian process. His work created a method for uncertainty quantification revolving around the use of dropout in networks and weight decay. This work was implemented in the command line tool for use in analyzing how certain a network is of a prediction for a new data point.

Given a new data point and a trained model we pass the point through the network repeatedly giving various predictions. The trained network is set to *model.eval()*. This setting changes the behavior of the network, such modules as *Pytorch Dropout* or *Pytorch BatchNorm1d* are essentially "turned off" since they are meant to be used only for the training process. The figure 5 example shows training error much higher than test, which is confusing. In that case the training error was calculated without *model.eval()* on. The test error is calculated separately in a function where *model.eval()* is enabled.

Continuing with the uncertainty calculation, the gradients are disabled so that information from the new point does not affect the state of the network. At this point the behavior of modules has been changed and gradients are frozen. However, the *Pytorch Dropout* modules specifically, are set to *model.training()* to continue their behavior of disconnecting weights with their fixed probability. Repeating this a thousand times for the point provides various predictions for the data point. This process allows the calculation of the predictive mean and its uncertainty in the predictive mean. Using our parameter knowledge of the network from design choices, we calculate the Gaussian process precision τ. The formula is,

$$\tau = \frac{l^2 p}{2N\lambda}$$
$$where\ l = prior\ length\ scale$$
$$p = 1 - p_{dropout}$$
$$N = number\ of\ training\ observations$$
$$\lambda = weight\ decay\ .$$

Taking the predictive variance vector we add τ^{-1} elementwise giving the uncertainty in the predictive mean. Using *ReLU* as the nonlinear activation function gives a less overly optimistic uncertainty than use of *Tanh*. To be conservative, the length parameter l used in the command line tool is the length full width at one-tenth of maximum of the Gaussian. For a tighter uncertainty another length parameter option is available- commented out in the code.

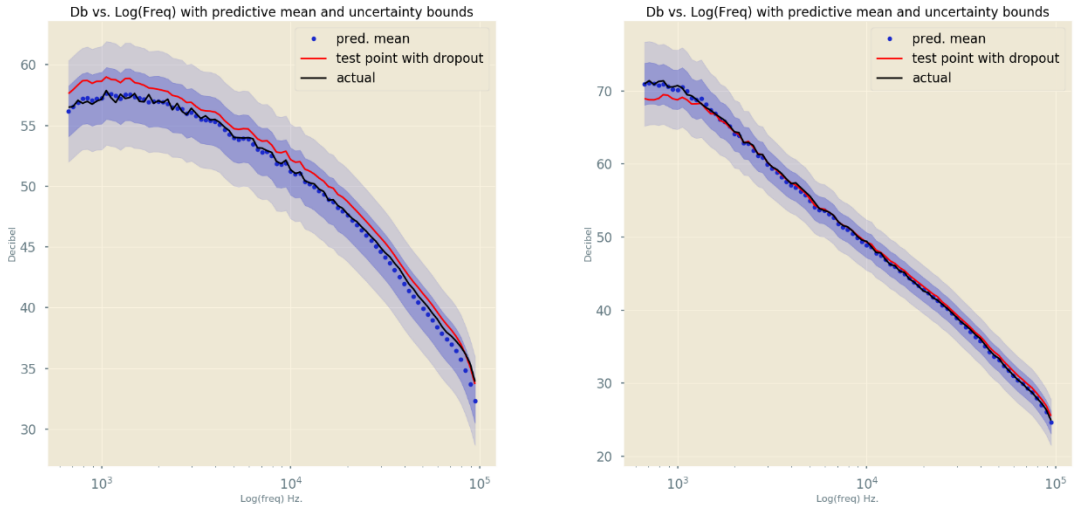


Figure 3. Uncertainty quantification. *Demonstrations of output plots from the command line tool for uncertainty of predictive mean. The light blue is 2 standard deviations, dotted line is the mean over 1000 repeated predictions of an input point, red is a randomly selected prediction from the set of 1000, and black is the actual. These plots were generated from "leftover" data not in train or test sets, and never seen at any point in the training process.*

II. Results

Recognizing the fact that we must utilize the *Pytorch* modules *ReLU* and *Dropout* for a best implementation of the uncertainty calculation focus a bit more on the *ReLU* results. The *Tanh* results are included for comparison. With the *Tanh* there are added complexities in the training process. Batch normalization is required to mitigate the effects of disappearing/exploding gradients. So at each layer the previous outputs must be processed to keep their input to the next layer near the linear section of the tanh curve. While this presents benefits for stopping mean propagation to the next layer, ReLU may have an advantage here. Since ReLU is calculated as $\max_{x \in X}(0, x)$, the mean activation is shifted to values greater than zero. With minmax scaling from 0 to 1 this may have worked in favor of ReLU as nonlinearity. If left to train for a high number of epochs, *Tanh* trains to overfit with extremely low MSE.

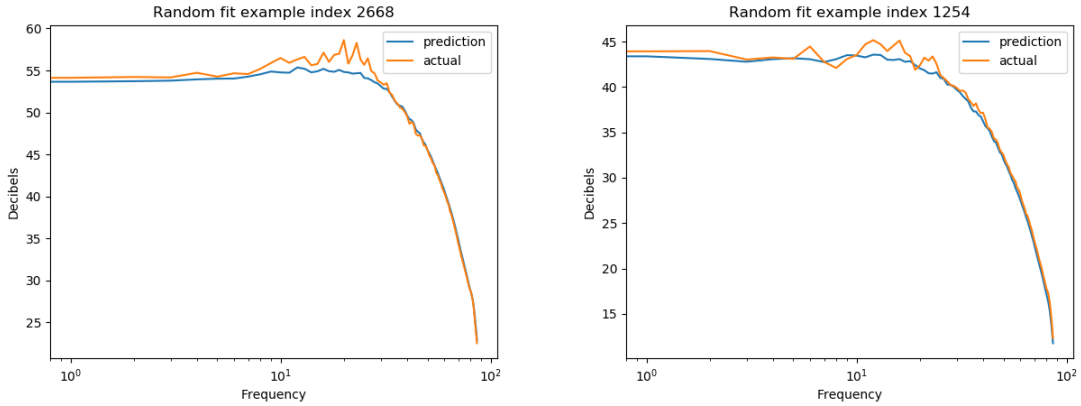


Figure 4. Random examples. *Both plots are random examples of predictions on the test set. Left is from configuration 2-0, rectangular nozzle aspect ratio 2:1 shielded by surface.*

A. Choosing the architecture

Results in the tables show the MSE averaged over 5 runs per configuration. Each configuration was run with 500 epochs. We test fixed learning rates of 0.1, 0.01, 0.001, and lastly we run models with weight decay enabled. This is ridge regression and decays to 1e-5. The models using feature engineering are marked with an asterisk.
We can see that DNN performs strongly at fixed learning rate of 0.01. DNN also performs strongly with weight decay enabled. It would seem that when learning rates are around 0.001 the DNN models can be caught in local minima. Dropout can be helpful with this as it provides noise beyond that provided by mini-batching. Considering the irreducible error, to go below MSE of 0.5 can be considered overfitting.

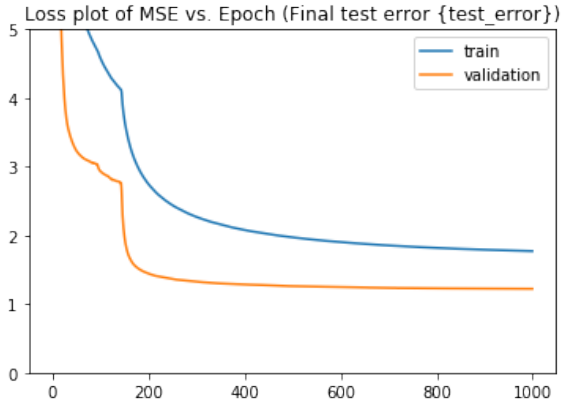


Figure 5. Wavy. *Example of escaping local minima*

For the final network implemented in the command line code we use the DNN with weight decay of 1e-5, early stopping when validation reaches MSE of 0.8, and *Dropout* modules with p=0.1. The DNN is trained using k-folds validation to further counter any overfitting effects. Some bonuses of this is that k-folds validation can make better use of smaller datasets, and give more views of under-represented data. K-folds is ran with k=5 folds, meaning the data for a configuration is randomly split into test and training set for 5 training folds/sessions. The network model retains its learned state from each fold, while the optimizer is reinitialized at each fold. So a new random train/test set is shown with a reset optimizer state per fold. Although the training is defaulted to run the full 20,000 epochs, it will stop early when validation MSE=0.8. This means that each fold will be shorter as the network trains up. The optimizer used is the *Pytorch Adam* optimizer with *MSELoss* as a cost function.

These design choices balance the "complexity capacity" of the network to handle future data, against anti-overtraining tools. Table 9 demonstrates how far down in MSE the DNN can train with feature engineering. Also in table 9 FF ReLU without feature engineering seems to be overtraining as its MSE on test set is increasing compared to table 7. The region plots in sections below show the spread of mean absolute error across the 87 predicted frequencies. The somewhat consistent variance for DNN versus FF suggests that, with early stopping, the model variance would generalize well by design. The hope is that this will prove useful for at least the next few years as researcher generate new data.

Table 9. FF vs. DNN, weight_decay=1e-5 (Ridge) @ 2000 epochs.

Config	FF ReLU	FF ReLU*	DNN1 ReLU	DNN ReLU*
1-1	0.84	0.30	0.18	0.14
1-0	1.33	0.46	0.23	0.17
2-1	1.13	0.42	0.24	0.17
2-0	2.18	0.84	0.47	0.36
4-1	1.24	0.62	0.35	0.23
8-1	1.49	0.59	0.34	0.23
8-0	2.59	1.94	0.45	0.27

B. Feature Selection

This section is relatively stand-alone compared to previous sections which are meant to answer the "Why?" of design choices for the command line tool. Here we look at which features were chosen by the FF and DNN networks under five repeated network initializations (varied random weight initialization) and varied train/test splits. The LASSO regularizer was applied to the output weights with a factor of 1.0. The lists below provide the set intersection across all configurations for the FF network, and similarly for the DNN. The last list is the intersection of FF and DNN unions.

An exhaustive break down of the selected features was not finished. Further work to examine these selections could prove interesting, in that they may show why or if these features are physically useful. Across many runs of the network it was notable that $\{angle^4, xE^2hE^2\}$ were present in almost every run. It seems as though there is some core set of features used and that each configuration will differ by a few features from the rest.

```
FF
#intersection
{'TSRangleangleangle','TTRangleangleangle','angleangleangle','angleanglean
gleangle'}
#union
{'Cos_Ma','Cos_Mj','Cos_NPR','Cos_TSR','Cos_TTR',
'Cos_angle','Cos_xE','Log_TSR','Log_TTR','Log_angle','Log_hE',
'Log_xE','Maangleangleangle','Mjangleangleangle','NPRangleangleangle',
'Sine_NPR','Sine_TSR','Sine_TTR','Sine_hE','Sqrt_TSR','Sqrt_TTR',
'Sqrt_angle','Sqrt_hE','TSR','TSRTSR','TSRTSRTSR','TSRTSRTSRTSR',
'TSRangleangleangle','TTR','TTRTSR','TTRTSRTSR','TTRTSRTSRTSR',
'TTRTTR','TTRTTRTSR','TTRTTRTSRTSR','TTRTTRTTR','TTRTTRTTRTSR',
'TTRangleangleangle','Tan_NPR','Tan_TSR','Tan_angle','angleangleangle',
'angleangleangleangle','xEangleangleangle','xEhE','xEhEMaangle','xEhEMjang
le','xEhENPRangle','xEhETSR','xEhETSRTSR','xEhETSRangle','xEhETTR','xEhETT
RTSR','xEhETTRTTR','xEhETTRangle','xEhEangle','xEhEangleangle','xExEhE','x
ExEhETSR','xExEhETTR','xExEhEangle','xExEhEhE','xExExEhE'}

DNN
#intersection
{'Cos_angle','Sine_angle','angleangleangleangle','hEMaangleangle',
'hEMjangleangle','hENPRangleangle','hEangleangleangle','xEhEangleangle','x
ExEhE','xExEhEMa','xExEhEMj','xExEhENPR','xExEhETSR','xExEhETTR','xExEhEan
gle','xExEhEhE','xExExEhE'}
#union
{'Cos_angle','Log_TTR','Log_xE','Maangleangleangle','Mjangleangleangle',
'NPRangleangleangle','Sine_TTR','Sine_angle','Sqrt_TTR','Sqrt_hE',
'TSRangleangleangle','TTR','TTRTTR','TTRTTRTTR','TTRTTRTTRTTR',
'TTRangleangleangle','Tan_TTR','angleangleangle','angleangleangleangle',
'hE','hEMaangleangle','hEMjangleangle','hENPRTSRTSR','hENPRangleangle',
'hETSR','hETSRTSR','hETSRTSRTSR','hETSRangleangle','hETTR','hETTRTSR',
'hETTRTSRTSR','hETTRTTR','hETTRTTRTSR','hETTRTTRTTR','hETTRangleangle',
'hEangleangle','hEangleangleangle','xEangleangleangle','xEhE',
'xEhEMaangle','xEhEMjangle','xEhENPRangle','xEhETSR','xEhETSRTSR',
'xEhETSRangle','xEhETTRTSR','xEhETTRTTR','xEhETTRangle','xEhEangle',
'xEhEangleangle','xEhEhE','xEhEhEMa','xEhEhEMj','xEhEhENPR','xEhEhETSR',
'xEhEhETTR','xEhEhEangle','xExEhE','xExEhEMa','xExEhEMj','xExEhENPR',
'xExEhETSR','xExEhETTR','xExEhEangle','xExEhEhE','xExExEhE','xExExExE'}

#intersection of ff and dnn
{'Cos_angle','Log_TTR','Log_xE','Maangleangleangle','Mjangleangleangle',
'NPRangleangleangle','Sine_TTR','Sqrt_TTR','Sqrt_hE','TSRangleangleangle',
```

```
'TTR','TTRTTR','TTRTTRTTR','TTRangleangleangle','angleangleangle',
'angleangleangleangle','xEangleangleangle','xEhE','xEhEMaangle',
'xEhEMjangle','xEhENPRangle','xEhETSR','xEhETSRTSR','xEhETSRangle',
'xEhETTRTSR','xEhETTRTTR','xEhETTRangle','xEhEangle','xEhEangleangle',
'xExEhE','xExEhETSR','xExEhETTR','xExEhEangle','xExEhEhE','xExExEhE'}
```

Figure. 6. Features used. *The lists above are the features selected by the FF (top intersection and union) and DNN (bottom intersection and union). These features had an intensity greater than 0.5 on the last epoch of training, on the far-right column of a heatmap plot.*

The heatmaps below were used as a tool to examine the effect of LASSO. The intensity of different connections from the input layer are meant to indicate the networks level of use for the feature. In some exploratory test runs the intersection across all models which was $\{angle^4, xE^2hE^2\}$ and one other variable that may have been misleading, was used to train a network. The MSE was greater than ~80 for all configurations after over 100 epochs. Using the union selected from FF the MSE drops within 20 epochs to nearly 0.8. The union from DNN goes even lower almost immediately. Taking the set difference $DNN - FF$ caused certain regions of the MAE error in the region plots to blow up. Further exploration will be interesting for those who attempt it.

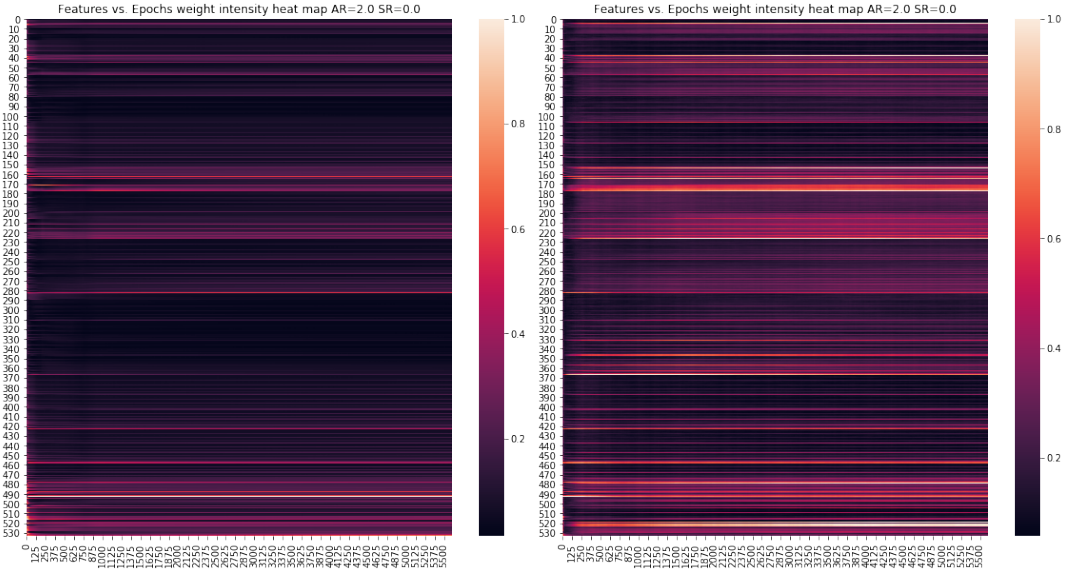


Figure 7. Connection intensity. *The above plots show the intensity of connection from each of 534 input variables to the 64 nodes in the first hidden layer. The streaks result from LASSO choosing a sparse basis from the input variables. Intensity per epoch is calculated as the sum of weights to a variable across a batch size of 50, then divided by the max value sum. Due to random initialization, some variables start with higher intensity and fade away if not useful to the network and vice versa.*

C. Variance Regions

The plots of mean absolute error and its variance provided below are meant to supplement the uncertainty plots provided by the command line tool. These plots illustrate the frequencies which prove to have higher error than others. This should be useful for consideration to a researcher using the tool for predictions. Furthermore, for future experiments this may help in design. The low frequency regions show higher variance in the predictions. Future work could attempt to reduce this.

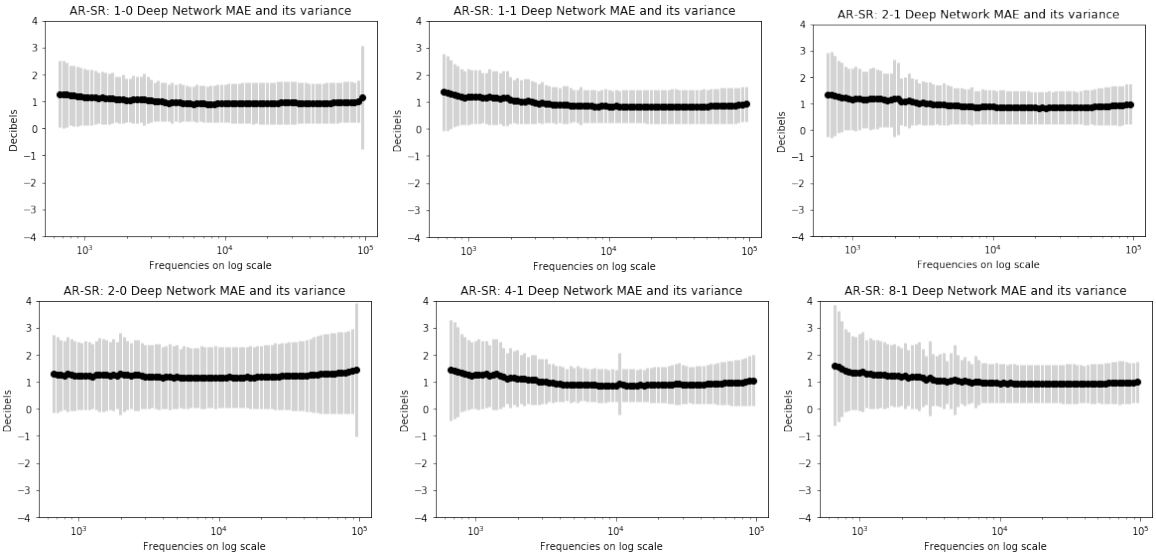
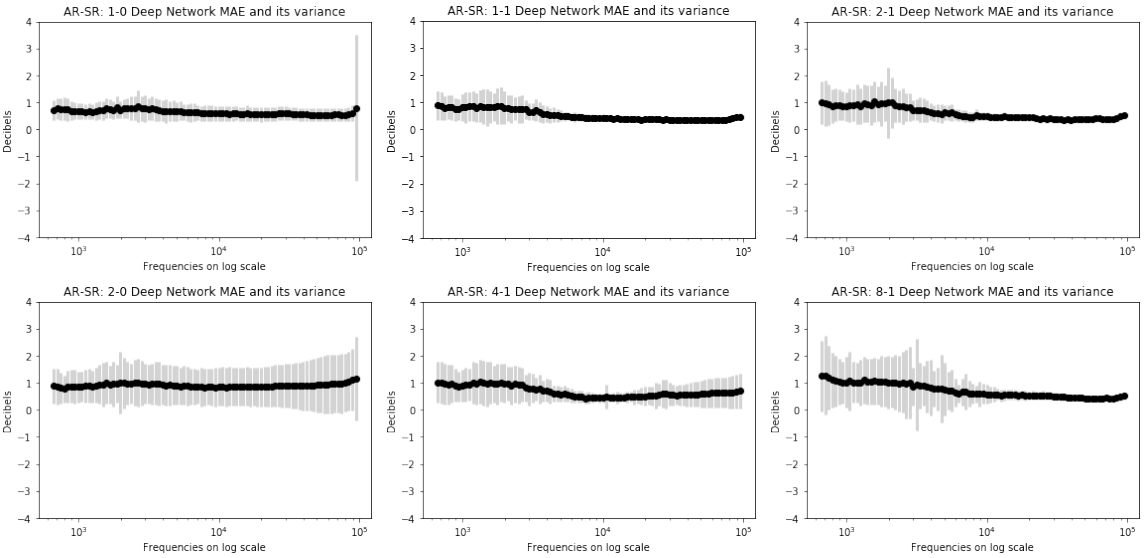


Figure 7. DNN regions of variance. *These are six of the seven configurations. The DNN has the variance in mean absolute error more consistent across the 87 predicted frequencies.*



III. Conclusion

Over the course of this project we expanded on previous work applying machine learning tools to jet acoustic data. In order to develop a command line machine learning based predictive tool we examined various architectures. Their capabilities on various subsets of the experiment dataset were evaluated. Considering possible future demands from the data a deep neural network design was arrived at. A command line tool was constructed and trained for use as an open source tool. In the process of this effort, the data was examined and various complications due to the complexity of artificial neural networks were weighed. Feature selection by the model was explored to open new lines of future inquiry.

Acknowledgments

The author thanks Chris Miller and Clifford Brown for performing the experiments and providing the data for this project. Additionally, the author would like thank Lauren MacIntyre for her project mentorship. This work was supported by the NASA Advanced Air Vehicle Program Commercial Supersonic Technology Project.

References

[1] Brown, C., "Jet-surface interaction test: far-field noise results," *ASME Turbo Expo 2012: Turbine Technical Conference and Exposition*, American Society of Mechanical Engineers, 2012, pp. 357–369.

[2] Brown, C. A., "Empirical Models for the Shielding and Reflection of Jet Mixing Noise by a Surface," *21st AIAA/CEAS Aeroacoustics Conference*, 2015, p. 3128.

[3] Brown, C., "An Empirical Jet-Surface Interaction Noise Model with Temperature and Nozzle Aspect Ratio Effects," 53rd AIAA Aerospace Sciences Meeting, 2015.

[4] Brown, C., "Including Finite Surface Span Effects in Empirical Jet-Surface Interaction Noise Models," 54rd AIAA Aerospace Sciences Meeting, 2016.

[5] Tenney, A. S., Glauser, M. N., and Lewalle, J., "A Deep Learning Approach to Jet Noise Prediction," *2018 AIAA Aerospace Sciences Meeting*, 2018, p. 1736.

[6] Gal, Y., Gharhamani, Z., "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning," arXiv:1506.02142, 2015

[7] Dowdall, J., "Machine Learning for Noise Modeling in Jet-Surface Interaction," NASA Glenn Research Center, Cleveland, OH, 2018

Appendix A

Data from 1/3 octave bands was taken using DNN with tanh nonlinearity and batch normalization for 3000 epochs. The LASSO regularization penalty factor was set to 1.0. The band frequencies present in the data are provided in brackets. The set of features selected by LASSO are provided in curly braces.

DNN with tanh nonlinearity and batchnorm1d @ 3000 epochs, LASSO regularization factor 1.0

One-third Octave Subset ['749.9', '794.3', '841.4']

{165, 493, 529, 530, 499, 531, 501, 533, 503, 500}
-> 1-1 average error 0.40
{493, 529, 530, 531, 500, 533, 499, 503, 501}
-> 1-0 average error 0.37
{493, 529, 530, 531, 500, 501, 499, 503}
-> 2-1 average error 0.59
{514, 163, 488, 492, 493, 529, 530, 531, 499, 533, 500, 503}
-> 2-0 average error 0.48
{493, 529, 530, 499, 500, 531, 501, 503}
-> 4-1 average error 0.64
{493, 529, 530, 531, 499, 501, 500, 503}

-> 8-1 average error 0.74
{493, 529, 530, 499, 500, 531, 533, 503, 501}
-> 8-0 average error 0.80

One-third Octave Subset ['891.3', '944.1', '1000.0', '1059.0', '1122.0']

{163, 488, 493, 529, 498, 499, 500, 530, 531, 503, 533, 502}
-> 1-1 average error 0.43
{163, 488, 492, 493, 529, 530, 531, 499, 533, 500, 503, 501}
-> 1-0 average error 0.44
{493, 529, 530, 499, 500, 531, 503}
-> 2-1 average error 0.52
{163, 165, 488, 492, 493, 45, 173, 174, 529, 530, 499, 500, 531, 533, 503, 175, 176, 532, 478}
-> 2-0 average error 0.54
{493, 529, 530, 531, 500, 499, 501, 503}
-> 4-1 average error 0.68
{493, 173, 529, 530, 499, 500, 501, 531, 503}
-> 8-1 average error 0.76
{493, 529, 530, 531, 500, 499, 501, 503, 533}
-> 8-0 average error 0.78

One-third Octave Subset ['1189.0', '1259.0', '1334.0']

{514, 493, 529, 530, 499, 531, 500, 498, 503, 502}
-> 1-1 average error 0.44
{493, 529, 530, 531, 499, 533, 500, 503, 501}
-> 1-0 average error 0.35
{493, 529, 530, 499, 500, 531, 498, 503}
-> 2-1 average error 0.67
{493, 529, 530, 531, 500, 533, 499, 503}
-> 2-0 average error 0.69
{493, 529, 530, 499, 500, 501, 531, 503}
-> 4-1 average error 0.69
{493, 529, 530, 531, 499, 500, 503}
-> 8-1 average error 0.93
{493, 529, 530, 531, 499, 533, 500, 503, 501}
-> 8-0 average error 0.76

One-third Octave Subset ['1413.0', '1496.0', '1585.0', '1679.0', '1778.0']

{493, 529, 530, 499, 500, 531, 498, 503, 501}
-> 1-1 average error 0.44
{493, 529, 530, 531, 499, 500, 501, 503, 533}
-> 1-0 average error 0.36
{493, 529, 530, 531, 499, 500, 503}
-> 2-1 average error 0.73
{493, 529, 530, 531, 500, 499, 533, 503}
-> 2-0 average error 0.81

{529, 530, 531, 499, 500, 503}
-> 4-1 average error 0.72
{493, 529, 530, 499, 500, 531, 503}
-> 8-1 average error 0.86
{529, 530, 531, 499, 500, 503}
-> 8-0 average error 0.85

One-third Octave Subset ['1884.0', '1995.0', '2113.0']

{493, 529, 530, 531, 499, 500, 503}
-> 1-1 average error 0.39
{514, 493, 529, 530, 499, 500, 501, 531, 503, 533, 532}
-> 1-0 average error 0.32
{493, 529, 530, 531, 499, 501, 498, 503, 500}
-> 2-1 average error 0.71
{493, 529, 530, 531, 500, 533, 499, 503}
-> 2-0 average error 0.94
{493, 529, 530, 531, 499, 500, 503}
-> 4-1 average error 0.68
{529, 530, 531, 499, 500, 503}
-> 8-1 average error 0.93
{493, 529, 530, 499, 500, 531, 533, 503}
-> 8-0 average error 0.88

One-third Octave Subset ['2239.0', '2371.0', '2512.0', '2661.0', '2818.0']

{493, 529, 530, 499, 531, 500, 503}
-> 1-1 average error 0.33
{493, 529, 530, 499, 531, 533, 500, 503}
-> 1-0 average error 0.38
{493, 529, 530, 531, 499, 500, 503}
-> 2-1 average error 0.63
{493, 529, 530, 499, 500, 531, 533, 503}
-> 2-0 average error 1.00
{493, 529, 530, 499, 500, 531, 503}
-> 4-1 average error 0.75
{493, 529, 530, 531, 499, 500, 503}
-> 8-1 average error 0.97
{493, 529, 530, 499, 500, 531, 533, 503}
-> 8-0 average error 0.93

One-third Octave Subset ['2985.0', '3162.0', '3350.0', '3548.0']

{163, 488, 492, 493, 529, 530, 499, 531, 533, 503}
-> 1-1 average error 0.30
{514, 163, 488, 493, 529, 530, 499, 531, 533, 500, 503, 501}
-> 1-0 average error 0.37
{493, 529, 530, 499, 500, 531, 498, 503}

-> 2-1 average error 0.45
{493, 529, 530, 499, 500, 531, 533, 503}
-> 2-0 average error 0.85
{493, 529, 530, 499, 500, 531, 501, 503}
-> 4-1 average error 0.58
{493, 529, 530, 531, 499, 500, 503}
-> 8-1 average error 0.94

One-third Octave Subset ['3758.0', '3981.0', '4217.0']

{514, 163, 493, 529, 530, 531, 499, 500, 533, 503, 498}
-> 1-1 average error 0.23
{514, 163, 516, 488, 492, 493, 529, 530, 499, 500, 531, 532, 503, 533}
-> 1-0 average error 0.38
{493, 529, 530, 499, 531, 500, 503}
-> 2-1 average error 0.37
{493, 529, 530, 499, 531, 533, 498, 503, 500, 532}
-> 2-0 average error 0.85
{493, 529, 530, 499, 500, 531, 502, 503}
-> 4-1 average error 0.43
{493, 529, 530, 499, 500, 531, 503}
-> 8-1 average error 0.80
{493, 529, 530, 531, 499, 533, 503}
-> 8-0 average error 0.93

One-third Octave Subset ['4467.0', '4732.0', '5012.0', '5309.0', '5623.0']

{493, 529, 530, 499, 500, 531, 498, 503}
-> 1-1 average error 0.18
{493, 529, 530, 531, 499, 533, 500, 503}
-> 1-0 average error 0.30
{493, 529, 530, 499, 500, 531, 502, 503, 501}
-> 2-1 average error 0.39
{163, 488, 492, 493, 529, 530, 499, 500, 531, 533, 503, 498, 532}
-> 2-0 average error 0.82
{493, 529, 530, 499, 500, 531, 503}
-> 4-1 average error 0.37
{493, 529, 530, 499, 500, 531, 503}
-> 8-1 average error 0.86
{493, 529, 530, 531, 499, 500, 533, 503}
-> 8-0 average error 1.32

One-third Octave Subset ['5957.0', '6310.0', '6683.0', '7079.0']

{514, 493, 529, 530, 499, 500, 531, 503}
-> 1-1 average error 0.16
{493, 529, 530, 531, 500, 533, 503}
-> 1-0 average error 0.29

{493, 529, 530, 499, 531, 500, 502, 503}
-> 2-1 average error 0.25
{163, 488, 493, 529, 498, 499, 500, 530, 531, 503, 533}
-> 2-0 average error 0.75
{172, 493, 529, 530, 499, 531, 500, 502, 503}
-> 4-1 average error 0.35
{493, 529, 530, 499, 531, 503}
-> 8-1 average error 0.60
{493, 529, 530, 531, 499, 503}
-> 8-0 average error 0.84

One-third Octave Subset ['7499.0', '7943.0', '8414.0']

{514, 163, 493, 529, 530, 499, 531, 500, 533, 503}
-> 1-1 average error 0.09
{163, 488, 492, 493, 529, 530, 531, 500, 532, 533, 503}
-> 1-0 average error 0.23
{514, 493, 529, 530, 531, 499, 500, 503}
-> 2-1 average error 0.22
{163, 488, 492, 493, 529, 498, 499, 530, 531, 533, 503}
-> 2-0 average error 0.72
{163, 488, 493, 529, 530, 499, 531, 533, 503}
-> 4-1 average error 0.29
{493, 529, 530, 499, 531, 503}
-> 8-1 average error 0.49
{493, 529, 530, 531, 499, 533, 503}
-> 8-0 average error 0.84

One-third Octave Subset ['8913.0', '9441.0', '10000.0', '10590.0', '11220.0']

{514, 163, 493, 529, 530, 531, 499, 533, 503}
-> 1-1 0.08
{493, 529, 530, 531, 533, 503}
-> 1-0 0.28
{163, 488, 492, 493, 529, 530, 499, 531, 500, 533, 503}
-> 2-1 0.17
{514, 163, 488, 492, 493, 529, 530, 499, 531, 533, 500, 503, 532}
-> 2-0 0.65
{493, 529, 530, 499, 531, 533, 503}
-> 4-1 0.25
{163, 493, 529, 530, 531, 499, 503}
-> 8-1 0.43
{493, 529, 530, 499, 500, 531, 533, 503}
-> 8-0 0.85

One-third Octave Subset ['11890.0', '12590.0', '13340.0']

{514, 493, 529, 530, 531, 499, 503}
-> 1-1 0.06
{493, 529, 530, 531, 503}
-> 1-0 0.25
{163, 488, 493, 529, 530, 499, 531, 533, 503}
-> 2-1 0.12
{163, 488, 492, 493, 529, 530, 499, 531, 533, 503}
-> 2-0 0.66
{493, 529, 530, 499, 531, 503}
-> 4-1 0.25
{493, 529, 530, 531, 499, 503}
-> 8-1 0.28
{493, 529, 530, 531, 499, 503}
-> 8-0 0.70

One-third Octave Subset ['14130.0', '14960.0', '15850.0', '16790.0', '17780.0']

{514, 493, 529, 530, 531, 533, 503}
-> 1-1 0.05
{493, 529, 530, 531, 533, 503}
-> 1-0 0.27
{514, 493, 529, 530, 499, 531, 503}
-> 2-1 0.11
{163, 423, 488, 492, 493, 529, 530, 531, 499, 533, 532, 503, 478}
-> 2-0 0.62
{493, 529, 530, 531, 499, 501, 503}
-> 4-1 0.23
{163, 488, 492, 493, 529, 530, 499, 531, 533, 503}
-> 8-1 0.30
{163, 488, 493, 529, 530, 499, 531, 533, 503}
-> 8-0 0.60