NASA/TP—2020-220385

Genetic Algorithm for Optimization of Neural Networks for Bayesian Inference of Model Uncertainty

Oscar M. Youngquist and Lauren P. McIntyre
Glenn Research Center, Cleveland, Ohio

April 2020

NASA/TP—2020-220385

Genetic Algorithm for Optimization of Neural Networks for Bayesian Inference of Model Uncertainty

Oscar M. Youngquist and Lauren P. McIntyre
Glenn Research Center, Cleveland, Ohio

National Aeronautics and
Space Administration

Glenn Research Center
Cleveland, Ohio 44135

April 2020

Acknowledgments

I would like to thank Clifford Brown and Christopher Miller of the Acoustics Branch at the NASA Glenn Research Center for supporting this work as subject matter experts in acoustics and providing the training and evaluation data. I would also like to thank Calvin Robinson and Herbert Schilling of the Scientific Computing and Visualization team at NASA Glenn for supporting this project and giving me the opportunity to perform research this summer. This work was supported by the NASA Advanced Air Vehicle Program Commercial Supersonic Technology (CST) Project.

This work was sponsored by the Advanced Air Vehicle Program
at the NASA Glenn Research Center

Contents

Genetic Algorithm for Optimization of Neural Networks for Bayesian Inference of Model Uncertainty

Oscar M. Youngquist[*] and Lauren P. McIntyre
National Aeronautics and Space Administration
Glenn Research Center
Cleveland, Ohio 44135

Summary

The objective of this work was to develop a genetic optimization algorithm that can design a neural network capable of producing uncertainty estimates along with predictions. This algorithm is necessary because the inclusion of uncertainty modeling in a neural network greatly complicates the network's design space, making the development of a converging model extremely difficult and time consuming. The genetic algorithm presented in this work uses a number of value ranges for various configurable neural network parameters to create a randomly generated population of network architectures. The initially generated population is then evolved over the course of several generations, with the best performing models breeding to produce novel network configurations. Mutations are randomly applied to the network designs to facilitate the development of adaptations beneficial to the task being performed. An experiment was conducted to validate the proposed algorithm, in which the genetic optimizer was tasked with producing a neural network capable of predicting the sound pressure level (SPL) resulting from jet-surface interaction (JSI) noise. The data used for this task was generated at the NASA Glenn Research Center in the Aero-Acoustic Propulsion Laboratory. Starting with an initial population size of 35 randomly generated networks, and evolved over the course of 10 generations, the genetic algorithm produced a design able to predict SPL as a result of JSI noise within 0.272 dB, on average.

Acronyms

CSV	comma-separated value
GPU	graphics processing unit
JSI	jet-surface interaction
MAE	mean absolute error
MSE	mean squared error
SPL	sound pressure level in decibels (dB)

[*]Summer Intern in Lewis' Educational and Research Collaborative Internship Project (LeRCIP). Undergraduate at Rose-Hulman Institute of Technology.

Introduction

In recent years, deep learning has revolutionized the use of artificial intelligence tools in fields such as biology, medicine, physics, and manufacturing (Refs. 1 to 3). Artificial neural networks, convolution, dropout, and other deep learning tools have emerged as the state of the art in regression and classification tasks. However, a shortcoming in traditional deep learning methods is the inability to capture model uncertainty. This can be a prohibitive factor in the adoption of deep learning techniques in many areas of research, including the fields listed above. This has led some researchers to adopt Bayesian inference techniques as an alternative (Refs. 4 to 6).

Bayesian probability theory offers researchers mathematically grounded techniques capable of performing regression and classification tasks, as well as the ability to reason about model uncertainty (Ref. 7). Despite this advantage, Bayesian modeling usually comes with a prohibitive computation cost attached. Therefore, researchers have sought the development of a deep learning framework capable of estimating model uncertainty.

To meet this demand, Yarin Gal introduced the mathematical basis for equating the use of dropout in neural networks to a Bayesian approximation of the well-established Gaussian process probabilistic model in his 2015 paper (Ref. 7). In short, this technique is accomplished by using dropout layers in a neural network to introduce random noise into the model's predictive process. Then, if several predictions are made on the same input, the model's predictions will fit a Gaussian distribution. From this distribution, one can calculate the input's predictive mean (the average of the predictions) and predictive uncertainty (the variance of the predictions). While this technique allows deep learning models to produce uncertainty estimates along with their predictions, it also introduces new complications in an already difficult design process.

Designing neural networks by hand—selecting the number of hidden layers, number of neurons, activation functions, the optimizer, and more—is extremely challenging. More often than not, engineers are forced to rely on intuition and experience rather than a fixed and well-understood design methodology. This is due to the vast, epistatic, noisy, and multimodal design space represented by a neural network's topology and hyperparameters (Ref. 8). It can take experts hundreds of hours to design, implement, evaluate, and optimize a neural network design. The introduction of noise into the model's predictions adds to the difficulty of this process by making it harder for neural networks to converge to an acceptably low loss value. Furthermore, this noise can also mask underlying issues in the design of the network itself, making it more difficult for engineers to discover cause and effect relationships in their models. Accordingly, the introduction of uncertainty estimation through dropout can further increase the time required to design neural networks.

This work introduces and experimentally validates a method to alleviate these challenges via a genetic algorithm that optimizes network design through an evolutionary process.

Algorithm Description

The method presented in this work uses a genetic algorithm to evolutionarily build, evaluate, and optimize the design of multilayer feed-forward neural networks for the Bayesian approximation of model uncertainty. This evolutionary process is accomplished by providing the algorithm with a dictionary of configurable neural network parameters and their associated acceptable values, as well as a few control variables. The algorithm uses these parameters to construct a population of randomly configured network designs that are evolved over the course of a prescribed number of generations. The implementation of this work is in Python and relies on the Keras API using the TensorFlow backend (Refs. 9 and 10). Additionally, this work was inspired, in part, by the DeepEvolve project developed by Jan Liphardt at Stanford University (Ref. 11).

Inputs and Outputs

There are 11 inputs for this algorithm: *N*, population_size, max_population_size, nn_param_choices, output_dir, mutate_chance, select_chance, *k*, epochs, training_data, and testing_data.

The *N* input is simply the number of generations over which the initial population of randomly generated networks will evolve. Population_size is the size of the initially generated population. For example, if this value is set to 50, then the first generation will have a population of 50 unique network designs. Likewise, max_population_size is the maximum size to which future generations are allowed to grow. This value is necessary because in this algorithm, the populations of future generations are allowed to grow in order to include potentially beneficial child networks.

The nn_param_choices input is a Python dictionary of configurable neural network design parameters and their corresponding acceptable values. The parameters available for configuration in nn_param_choices are as follows:

- nb_neurons: number of neurons in a layer (a distinct nb_neurons value is selected for each layer)
- nb_layers: number of hidden layers in the network
- activation: activation functions (the same activation is used for each layer)
- optimizer: list of potential optimizers
- lr: learning rate for the optimizer
- clipnorm: the clipping value used by the optimizer
- dropout_prob: dropout probability
- w_decay: the weight decay value
- batch_size: number of samples per batch

The value of output_dir determines the location where all the outputs generated by the algorithm will be saved, and mutate_chance is the percent likelihood a point mutation will occur in a network's design. The parameter *k* is the percentage of the networks that will be carried over to the next generation. If $k = 0.25$, then the top 25 percent, or 75th percentile, of networks—based on their fitness score—will be carried over to the next generation of networks automatically. Select_chance is the percent likelihood a network not in the *k*th percentage will be carried over into the next generation. Finally, epochs is the number of epochs for which each network will be trained before being scored, and training_data and testing_data are simply the data on which the neural networks will be trained and evaluated.

There are two types of output from this algorithm. The first is a log file containing information about each generation of networks evaluated, namely, the generation's population size and the average performance across the population. Second is a folder that is generated for each network design that is evaluated. Each of these folders contains five additional files, one of which is an instance of the trained model, saved as an .h5 file type, that is associated with that folder. There are also two comma-separated value (CSV) files, one containing the model's training history, and the other containing values used to scale the network's inputs. Lastly, the folder contains two txt files, one describing the network's parameter configuration and one containing the network's evaluation score.

Evolutionary Algorithm

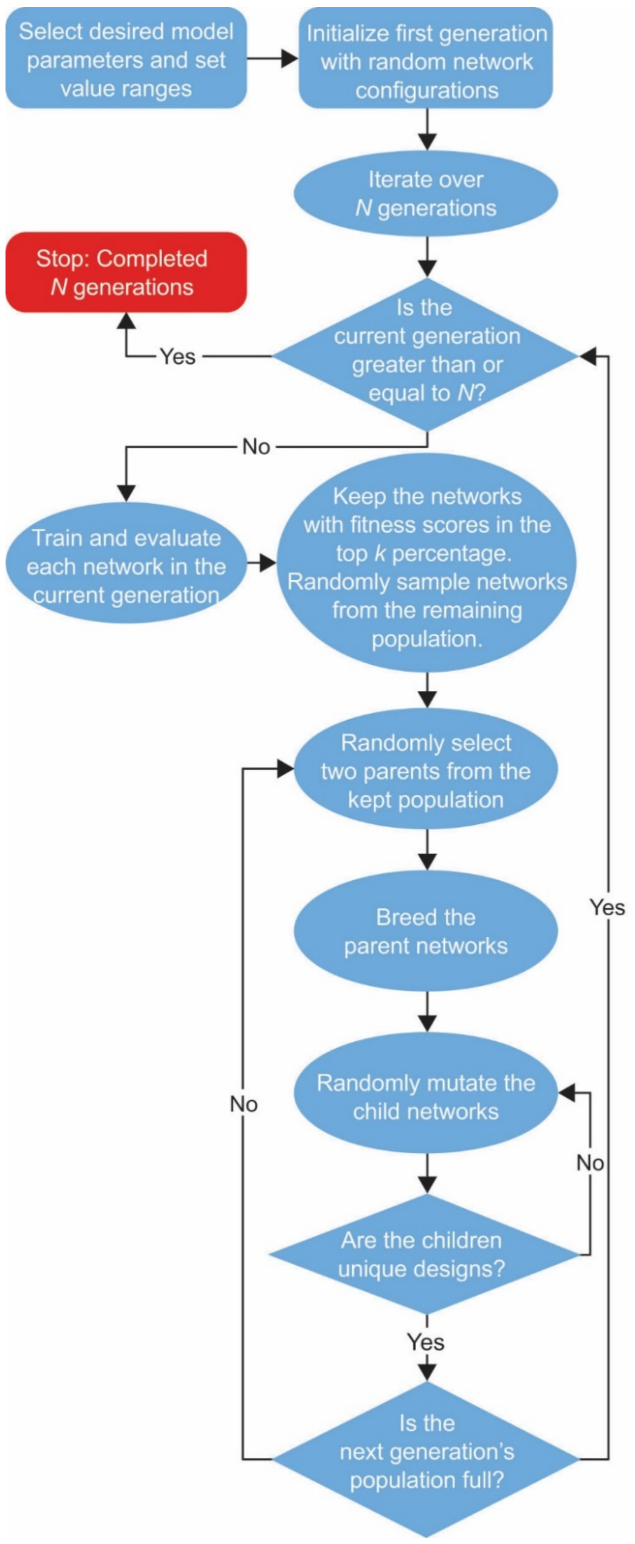
This section details the functional flow of the algorithm. An overview of this process can be seen in Figure 1. After providing the algorithm with the necessary inputs, the algorithm constructs the first generation's population by randomly selecting a value for each of the parameters from nn_param_choices for each candidate design. These values are then used to instantiate a neural network with the corresponding characteristics. After the entire population is created, the algorithm then evaluates each candidate design. The networks are trained one at a time, with each training on the user-provided training data for the number of epochs specified by the value of epochs. The size of the input and output layers is automatically calculated from the size of the input and target data. After training, each network is then evaluated on a held out test set, also provided by the user. The fitness of each network is calculated via mean squared error (MSE). The top *k*th percentage of these networks are selected to be carried over to the next generation. Additionally, the remaining networks in the population are iterated over and each network has a percent likelihood, predetermined by the value of select_chance, of also being selected to be carried over to the next generation. From the pool of carried-over networks, two designs at a time are randomly selected to be bred.

The breeding process involves the recombination of the two parent networks into two new child networks. Recombination, also called crossover, is the process in sexual reproduction in which entire sections of the parents' genomes are swapped. This leads to novel genetic configurations and is the main advantage of sexual reproduction. In our case, recombination is accomplished in the following steps:

1. One parameter is randomly selected as the "breakpoint" for recombination from the list of network parameters (as described in the Inputs and Outputs section).
2. For all the parameters up until the breakpoint, Child 1 receives all its parameters from Parent 1, and Child 2 from Parent 2.
3. From the breakpoint to the end of the parameter list, Child 1 receives all its remaining parameters from Parent 2, and Child 2 from Parent 1.

The process of recombination increases the likelihood that beneficial sequences of genes, in this case network parameters, from both parents will be incorporated into a single, better adapted genome.

After two new children are produced from the parent networks, each child is then randomly mutated. This mutation is random in that it is not a guarantee that the children will be mutated. The likelihood that each child will be mutated is governed by value of mutate_chance. Furthermore, if a child is mutated, the gene, or parameter, that is mutated is randomly selected. When a parameter is mutated, the value of that parameter is replaced by a randomly selected value from the list stored in nn_param_choices. Next, each child is checked for



Select desired model parameters and set value ranges
Initialize first generation with random network configurations
Iterate over N generations
Stop: Completed N generations
Yes
Is the current generation greater than or equal to N?
No
Train and evaluate each network in the current generation
Keep the networks with fitness scores in the top k percentage. Randomly sample networks from the remaining population.
Randomly select two parents from the kept population
Yes
Breed the parent networks
No
Randomly mutate the child networks
No
Are the children unique designs?
Yes
Is the next generation's population full?

Figure 1.—Flowchart representing the genetic algorithm presented in this work.

uniqueness. The algorithm keeps track of the network configuration of all evaluated networks, and the new children are checked against this list. This is done to avoid wasting time training and evaluating a configuration that has already been scored. If a child is found to be a duplicate, it will be randomly mutated until a unique design is generated.

The breeding process is continued until the next generation's population size is at least equal to that of the previous generation, but is still less than the value of max_population_ size. Any children created that exceed the max_population_size are discarded.

This entire process is repeated *N* times. After the *N*th iteration, the generation number, model number, score, and network configuration of the top five most fit networks are printed to the log file.

Experimental Validation

Dataset

The performance of this algorithm was evaluated on a regression task using a jet-surface interaction (JSI) noise dataset. This dataset was collected from experiments conducted in the Aero-Acoustic Propulsion Laboratory at the NASA Glenn Research Center (Ref. 12). The JSI dataset consists of 42,165 records, each with 18 features and 87 target values. The 18 features represent various design parameters of the nozzles of jet engines as well as assorted experimental conditions. The targets are the sound pressure level (SPL), or "loudness," in decibels of the JSI noise at 87 unique frequencies.

Experimental Setup

The goal of this regression task is to use the 18 features to predict the SPL at the 87 target frequencies. Domainexpertise-informed feature engineering was applied to the given features to produce a total of 66 features. The full dataset was split into 60 percent training, 20 percent validation, and 20 percent testing sets. The network parameter values supplied to the algorithm in this experiment are as follows:

1. nb_neurons: 16, 32, 64, 128, 256, 512, 768, 1,024
2. nb_layers: 1, 2, 3, 4
3. activation: relu, elu, tanh, sigmoid
4. optimizer: rmsprop, adam, sgd, adagrad, adadelta, adamax, nadam
5. lr: 0.0001, 0.001, 0.01, 0.1
6. clipnorm: 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0, 2.25, 2.5, 2.75, 3.0, 3.25, 3.5, 3.75, 4.0, 4.25, 4.5, 4.75, 5.0
7. dropout_prob: 0.005, 0.01, 0.05, 0.1
8. w_decay: 1×10^{-9}, 1×10^{-8}, 1×10^{-7}, 1×10^{-6}, 1×10^{-5}, 1×10^{-4}
9. batch_size: 16, 32, 64, 128, 256, 512

Additionally, *N* was set equal to 10, population_size equal to 35, and max_population size equal to 50. Lastly, mutate_chance was set to 0.2, select_chance was set equal to 0.1, *k* to 0.1, and epochs equal to 100. After the genetic optimization process, the top-performing model is retrained and evaluated on the same training, validation, and testing data subsets.

However, in this training phase the epoch count was not limited; the model would train until it reached a validation loss of 0.2 or below. This procedure was followed in order to fully evaluate the genetically constructed network architecture.

Results

Using the genetic algorithm, the duration of the experiment was 35 h, 45 min, and 32 s. The hardware used to perform the task was a NVIDIA® Tesla® (NVIDIA Corporation) P100 graphics processing unit (GPU) with 12 GB of memory. The five top-performing network configurations can be seen in Table I.

TABLE I.—TOP-PERFORMING GENETICALLY PRODUCED MODELS

Model	Layers[a]	activation	optimizer	learning rate	clipnorm	dropout_prob	w_decay	batch_size	MSE[b]
1	128, 512	sigmoid	adamax	0.01	2.25	0.005	1×10^{-8}	32	0.31
2	128, 512	sigmoid	adamax	.01	1.75	.005	1×10^{-6}	16	.32
3	128, 512	sigmoid	adamax	.01	2.75	.005	1×10^{-9}	16	.32
4	128, 512	sigmoid	adamax	.01	3.25	.005	1×10^{-8}	16	.32
5	128, 512	sigmoid	adamax	.01	2.25	.005	1×10^{-9}	16	.33

[a]Values for nb_neurons and nb_layers are condensed into Layers column.

[b]Mean squared error.

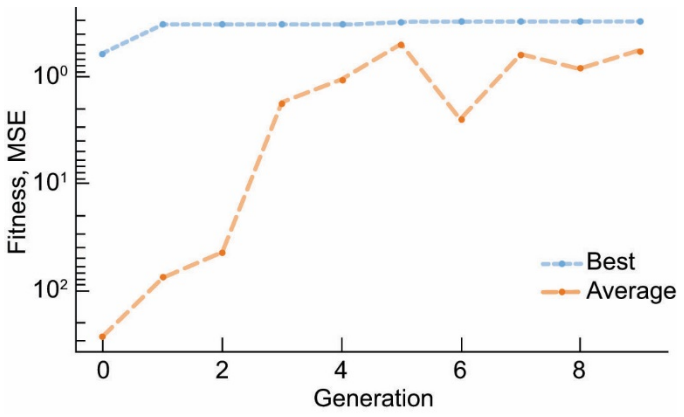


Figure 2.—Average fitness per generation and fitness of most fit model per generation; y-axis is inverted and log scaled. MSE, mean squared error.

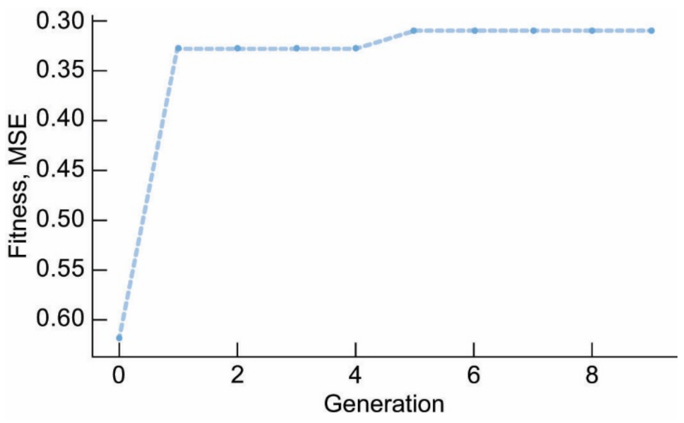


Figure 3.—Fitness of most fit model per generation. MSE, mean squared error.

Figure 2 shows the average fitness of each generation as well as the fitness of the most fit model per generation. Figure 3 shows the fitness of the most fit model per generation in isolation in order to show the progression of the most fit model in more detail.

One interesting observation that can be made from these results is that all of the top-performing networks have several parameters in common. Moreover, the structural layers parameter—the parameter that determines the network's "physical" architecture—is the same across all the top networks, with each network having two hidden layers with 128 and 512 neurons, respectively. Furthermore, all the networks have the same activation function, optimizer, learning rate, and dropout probability. Most likely this is due to the recombination approach to producing child networks during the evolution process. As stated in the Evolutionary Algorithm section, the purpose of recombination is to encourage large, epistatic adaptations from multiple configurations to merge into a single genome. Therefore, because all but one of the shared parameters occur sequentially in the genome, or network configuration dictionary, it is likely that one network with these traits was produced—likely in generation four, based on Figure 3—which was then recombined with an arbitrary number of other networks. Networks with these traits then performed so well that configurations that modified these values were almost always outperformed.

These findings suggest that the shared network parameters are some of the most important, if not the most important, parameters in determining the performance of a neural network when solving this problem. Likewise, the observation that the variation experienced between the top five models for the other parameters results in very little difference in the fitness of these models would suggest that these parameters are less important in determining the network's success.

Retraining and evaluating the top-performing network configuration (model 1 in Table I) results in an MSE score of 0.151 and a mean absolute error (MAE) score of 0.272 on the test set. The MAE score indicates that the network designed by the genetic optimization algorithm can predict JSI noise within 0.272 dB, on average. This accuracy falls well within the 0.5-dB tolerance of the experimental equipment of the Aero-Acoustic Propulsion Laboratory, making this model viable for design analysis purposes.

Conclusions

In this work, a genetic algorithm for the evolutionary optimization of a neural network for the Bayesian approximation of model uncertainty was introduced. This algorithm is necessary because the inclusion of uncertainty modeling in a neural network greatly complicates the network's design space, making the development of a converging model extremely difficult. An experiment was performed in which the genetic algorithm was tasked with producing network designs to predict the jet-surface interaction (JSI) noise of 87 frequencies based on 18 input features. After 10 generations with an initial population of 35 candidate designs, the genetic algorithm produced a design that was able to accurately predict JSI noise within 0.272 dB.

The experimental results found in this study demonstrate this algorithm's ability to produce highly successful, converging network designs in a complex and noisy design space. This algorithm has several potential benefits, not just in modeling JSI noise, but in any deep learning or machine learning project that requires uncertainty estimates. Therefore, this algorithm enables the wider adoption of neural networks in many fields. Furthermore, because this algorithm handles the network design process itself, a relatively novice user can take this tool and develop neural networks without being an expert in network design.

There are some potential drawbacks to the approach presented in this work, however. One potential disadvantage is that this algorithm in its current state only designs variations of simple multilayer feed-forward neural networks. While this type of network is capable of performing at state-of-the-art levels on most regression and classification tasks, highly specialized applications could require the use of convolutional or other neural network types. However, with fairly straightforward modifications, one could easily enable the algorithm to design any other style of neural network that is required. A second potential disadvantage is that genetic optimization makes no guarantee as to the global optimality of the designs it produces. Despite this, the genetic approach may be preferred because it is orders of magnitude faster than grid or random search methods and, on average, noticeably faster than Bayesian optimization routines (Ref. 13).

References

1. Ching, T., et al.: Opportunities and Obstacles for Deep Learning in Biology and Medicine. J. R. Soc. Interface, vol. 15, no. 141, 2018.
2. Pang, Long-Gang, et al.: An Equation-of-State-Meter of Quantum Chromodynamics Transition From Deep Learning. Nat. Commun., vol. 9, no. 210, 2018.
3. Wang, Jinjiang: Deep Learning for Smart Manufacturing: Methods and Applications. J. Manuf. Syst., vol. 48, 2018, pp. 144–156.
4. Herzog, Stefan; and Ostwald, Dirk: Experimental Biology: Sometimes Bayesian Statistics Are Better. Nature, vol. 494, 2013, p. 35.
5. Trafimow, David; and Marks, Michael: Editorial in Basic Applied Social Psychology. Basic Appl. Soc. Psychol., vol. 37, 2015, pp. 1–2.
6. Zdeborová, Lenka; and Krzakala, Florent: Statistical Physics of Inference: Thresholds and Algorithms. Adv. Phys., vol. 65, no. 5, 2016, pp. 453–552.
7. Gal, Yarin; and Ghahramani, Zoubin: Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. Cornell University, Ithaca, NY, 2015. arXiv:1506.02142v6 Accessed Feb. 6, 2020.
8. Miller, Geoffrey F.; Todd, Peter M.; and Hegde, Shailesh U.: Designing Neural Networks Using Genetic Algorithms. Proceedings of the Third International Conference on Genetic Algorithms, 1989, pp. 379–384.
9. Abadi, M., et al.: TensorFlow: A System for Large-Scale Machine Learning. Presented at the 12th USENIX Symposium on Operating Systems Design and Implementation, Savannah, GA, 2016, pp. 265–283.
10. Keras: The Python Deep Learning Library. 2015. https://keras.io Accessed Feb. 6, 2020.
11. Liphardt, Jan: GitHub Repo. DeepEvolve, 2017.
12. Brown, Clifford: Jet-Surface Interaction Test: Far-Field Noise Results. ASME GT2012–69639, 2012, pp. 357–369.
13. Mori, Naoki; Takeda, Masayuki; and Matsumoto, Keinosuke: A Comparison Study Between Genetic Algorithms and Bayesian Optimize Algorithms by Novel Indices. Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation, 2005, pp. 1485–1492.