Synthetic CFD Estimation for Blackhawk Airfoil Drag Coefficient

James Ross [*]
3909 Halls Ferry Rd. Vicksburg MS, 39180

Robert Haehnel[†]
72 Lyme Road, Hanover, New Hampshire, U.S.A, 03755

Luke D. Allen[‡]
72 Lyme Road, Hanover, New Hampshire, U.S.A, 03755

Brianna Thompson[§]
3909 Halls Ferry Rd. Vicksburg MS, 39180

Yi Zhang[¶]
3909 Halls Ferry Rd. Vicksburg MS, 39180

Stephen Wang[‖]
72 Lyme Road, Hanover, New Hampshire, U.S.A, 03755

Through understanding the effects of the modifications to the geometry of a rotorcraft blade airfoil, we will be able to better model airfoil performance. To analyze the performance of different airfoils, a morphing tool, Parfoil, is used to produce an airfoil geometry. The performance of the airfoil geometry is calculated using a 2D Computational Fluid Dynamics (CFD) model in the C81gen computer code; the airfoil performance, as a function of flow velocity and angle of attack, is saved in look-up tables called C81 tables or "air tables." The effects of airfoil modification on rotor blade performance are then assessed using a rotorcraft comprehensive analysis (CA) tool using updated air tables and successively refined with a numerical optimizer. Using CFD to calculate the air tables is the most computationally expensive part of evaluating rotorcraft performance with CA, and in an optimization simulation thousands of air tables need to be generated to determine optimal airfoil geometry. As an alternative to directly generating these air tables for each evaluation point with CFD, use of a machine learning (ML) surrogate model is explored in this work, and have the potential to significantly decrease analysis time by replacing the, on average, 2-hour long CFD calculation with a surrogate that computes and air table in less than a second. A base ML surrogate model was created to

[*]Computer Scientist, U.S. Army Engineer Research and Development Center
[†]Reseach Mechanical Engineer, U.S. Army Engineer Research and Development Center
[‡]Research Mechanical Engineer, U.S. Army Engineer Research and Development Center
[§]Computer Scientist, U.S. Army Engineer Research and Development Center
[¶]Computer Scientist, U.S. Army Engineer Research and Development Center
[‖]Computer Scientist, U.S. Army Engineer Research and Development Center

determine the feasibility of using a ML surrogate model to estimate the performance of an aircraft airfoil. The base ML surrogate model showed a respectable R^2 output for error from the loss function during training, but upon closer inspection during the validation step of training the initial network, many individual predictions still showed an unacceptable magnitude of error. The objective of this work is to improve the ML surrogate model's accuracy so that it becomes a feasible alternative for estimating the air tables that would ordinarily be produced by running CFD simulations. Because the computational cost of running CFD models for estimating the rotorcraft airfoil performance is prohibitive, having a ML surrogate model would allow for estimating a large number of potential airfoil designs in days rather than years. A ML surrogate model would also allow for a more efficient search of possible designs leading to a more robust final design. This final design could potentially reduce the number of flight tests that were previously required to find a similar design. The results defined in this work are an analysis of the effects of modifying the hyperparameters of the base ML surrogate model to improve accuracy and reduce computational effort.

I. Introduction

The starting point for this effort is an initial hand configured neural network which resulted in a network that suffered from overfitting IV.B. Starting with this base model we apply hyperparameter optimization to make an attempt at finding a more accurate neural network design. Furthermore, we note that the rotor performance metric being used by Allen et al. (the rotor power coefficient, C_p) is most sensitive to the airfoil drag. Therefore, this effort focuses on building a surrogate for computing the c_d portion of the air table; the same methodology can be readily applied to models that include estimation of c_l and c_m. In this section we will introduce the concepts and provide a small description of Linear Interpolation II, Machine learning II.A, Original CFD Surrogate Model Analysis II.B, and Hyperparameter Optimization II.C.

II. Linear Interpolation

Linear interpolation is commonly used for estimating new data points in between connecting pairs of known points determined from experiment or high fidelity modeling, e.g., CFD [1]. This operation is simple and fast, so it significantly reduces computational time when a large number of air tables are needed. Figure 1 shows that relative to the number of points originally computed to produce the interpolation database, time required to perform CFD runs increases linearly while time required for interpolation remains constant. In this example, the speedup of the interpolation method starts exceeding using only CFD by an order of magnitude at 10,000 tables, and by two orders of magnitude with 100,000 tables. However, the benefit of interpolation disappears with about 1,000 tables or less due to a base number of CPU hours required to generate the original database for interpolation; in the test case, 304 tables were created with CFD to facilitate linear interpolation.

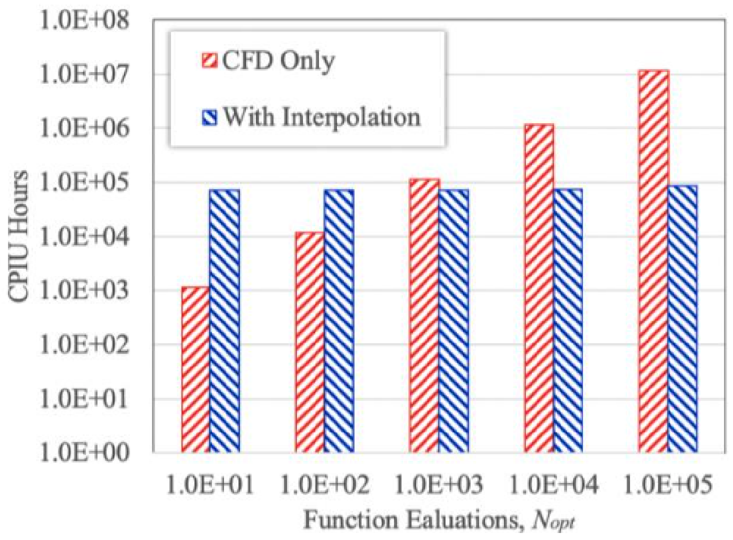


Fig. 1 CFD Compared with Interpolation

Though linear interpolation is computationally efficient, reasonably accurate (provided the grid spacing is not too large) and well behaved, linear interpolation to parameter space greater than 3 starts to become unwieldy [2, 3]. Furthermore, linear interpolation schemes are most efficient on a regularized grid structure. However, it is often advantageous to sample the parameter space in a non-uniform way, concentrating more points in areas of most interest while sparsely sampling regions that are perceived to be less fruitful. The efficiency of linear interpolation breaks down when nearest neighbor searches consume significant amounts of computational time on large, irregularly sampled parameter spaces. Alternative surrogate models, such as machine learning, can be more efficient on irregularly sampled parameter spaces.

A. Machine Learning

We are experiencing a revolution in the field of machine learning, which has been enabling advances across a wide range of scientific and engineering areas [4, 5]. Machine learning is focused on the development of algorithms with the ability to build models from data obtained without explicit use of mathematical models [6]. Applying machine learning methods to CFD is focused on the possibility of increasing the speed of highly accurate simulations, developing turbulence models with higher accuracy, and producing advanced models, beyond what can be achieved using classical approaches [7]. Some researchers have proposed ML methods for CFD and obtained good results, such as the turbomachinery design, turbulence modeling, and heat-transfer for aerodynamic optimization [7–9]. Among the machine learning methods is the use of a neural network, a popular supervised learning algorithm for modeling complicated systems [10–12]. Neural networks are based on a collection of connected nodes called neurons, which are arranged in parallel layers that are fully interconnected by weighted connection. The most important characteristic of a neural network is learning from data [13].

B. Original CFD Surrogate Model Analysis

A ML surrogate model, more specifically, a regression-based neural network (RNN), receives a dataset as input. Within the RNN there are many parameters which process the dataset, and these parameters are known as neurons. Each of these neurons is tuned and collectively determine a function for predicting data values. Neurons are positioned in a layered data structure. The first layer is known as the input layer, and the only job it has is to accept the incoming data source. Then, there are one or more layers known as hidden layers which are used to introduce non-linearity into the network. After hidden layers, there is a single layer known as the output layer, which provides the prediction of the network. Each layer contains an activation function within its neurons, which determines how the neurons are used in the calculation of the network's prediction. In summary, the configuration of the network's neurons, the number of layers, and the activation function within each neuron are used to process the data and provide a value to the user.

Our base surrogate ML model was designed manually with 16 neurons in the input layer, three hidden layers, and a single-neuron output layer. We soon realized this model was not sufficient for the dataset we provided; 31.8% of the data points had more than 10% error. The original model also performed better on the training dataset than on the test dataset which is commonly referred to as overfitting (see Section IV.B). This poor performance on test data but great performance on training data is due to the model learning too much about the training data.

C. Hyperparameter Optimization

Since the original surrogate model appeared to suffered from overfitting, we looked to improve on the base model through the use of hyperparameter optimization. Hyperparameter optimization is the process of sweeping through values for model parameters to find a set of parameters that produce an answer within an acceptable error range. Neural networks are not always accurate when making predictions, but could be described as producing an expert-level human like response to a question. Therefore, the goal of our use of hyperparameter optimization was to find a network that not only gave good quality of fit metrics (e.g., low RMSE and high R^2 values for the entire model), but was accurate in key areas of concern, and in this example, at very small drag coefficient values. Our search will seek to optimize hyperparameters important to the network design with conditions listed below:

- Number of hidden layers/neurons in each layer: How many parameters does the model have, and how deep is it? (Section III.D)
- Hidden layer activation: How is non-linearity introduced in the model? (Section III.E)

- Output layer activation: How should the model map the final result?(Section III.F)
- Optimizer: What learning optimizing algorithm should the model use? (Section III.G)
- Learning rate: What is the response level of the model to the estimated error of the output? (Section III.H)
- Epochs: How many times should the model be trained before it starts overfitting? (Section III.I)
- Loss function: How is model performance quantified? (Section III.J)

III. Neural Network Components and Selections

In a prior work, Allen et al. (2021) used linear interpolation to estimate the air tables from the database generated from CFD [14]. While interpolation is an efficient calculation, its accuracy depends heavily on how far apart each data point is from its paired point. Using a finely resolved regular grid of CFD runs would produce closely-spaced points, but is computationally expensive. As such, a practical CFD run would leave us with wider gaps between data points, potentially leading to greater inaccuracies in interpolated points that are far from a CFD computed point. Given this limitation, our objective shifted towards creating a ML surrogate model that can accurately reproduce points in the CFD created database, as well as accurately interpolate points for locations not contained in the database. Another requirement of the the ML model would be to compute required air tables at a rate several orders of magnitude faster than CFD. We were also looking for an improvement in accuracy of expected values in between points produced from CFD runs over the expected values produced by the linear interpolation method. In the following section we are going to describe the database, neural network libraries, hyperparameter optimization libraries, and the neural network components.

A. Dataset

The dataset is composed of air tables which are a compilation of lift, drag, and moment coefficients as a function of free stream Mach number (M) and angle of attack (AoA). Once these values for the air tables are generated using CFD, those values are provided to the ML model for training. In this work we only used the drag portion of the air tables to train a c_d surrogate model. In total, we generated 304 tables using CFD, which contained over 60,000 training points. Ninety percent of that data is allocated to the training set, which the regression-based neural network (RNN) uses to determine a non-linear function for predicting new air tables for new airfoil shapes. The remaining ten percent of the values within the air tables become the testing set, which is used for evaluating expected performance of the model on unseen data.

B. Neural Network Libraries

The Keras library was used for creating the neural network used in this work. Keras, which provides easy to use commands to interface with Google's TensorFlow library, provides a clean interface to build regression-based neural network (RNN) models [15]. Though Facebook's PyTorch is another popular choice, Keras was used since the team had more experience with this tool.

C. Hyperparameter Optimization Libraries

Because a regression-based neural network (RNN) has endless possible configurations, it is highly unlikely to find the optimal RNN by hand. To more efficiently explore the search space, we decided to use hyperparameter tuning libraries. These tools can automate testing of different RNNs based on hyperparameter conditions that we predetermine. Two such libraries that stood out were Optuna and KerasTuner. Both have a simple syntax to set up the optimization process and include state-of-the-art search algorithms. KerasTuner is integrated within Keras, so this library was the natural starting point. However, we found some of the functionality within Optuna to be helpful for this this work. For example, Optuna allows a user to store trials in a SQL database, allowing for organized viewing of past trials and checkpoints to resume optimization. By contrast, the KerasTuner storage format was less accessible and used several times more storage when tested for this effort. Pruning is another feature of Optuna that was helpful in performing hyperparameter optimization for this work. Pruning ends unpromising trials early based on a certain condition (e.g. if its intermediate value is worse than the median of past trials), which cuts down on optimization time. Because of efficient storage and trial execution, Optuna was chosen for hyperparameter tuning [16, 17]. Optuna has several search algorithms for determining hyperparameters. For hyperparameters with a limited search space (e.g. optimizer, activation function, and loss function), we used grid search to perform an exhaustive evaluation. For hyperparameters with an potentially large search space (e.g. number of layers and neurons in each layer, learning rate, epochs), we used Bayesian search, which employs probabilistic measures to narrow in on optimal regions.

D. Number of Hidden Layers and Neurons per Layer

For a regression-based neural network (RNN), the optimal number of hidden layers and neurons per hidden layer largely depends on the problem. Not having enough hidden layers/neurons can result in underfitting, which means the model learns too little about the training data to adequately make predictions. On the other hand, having too many neurons in the hidden layers may cause overfitting, where the model learns patterns in the training data so well that it is unable to generalize to unseen testing data. Most problems can see reasonable results with 1-5 hidden layers, depending on the trade-off one wants between accuracy and computation time [18]. Within the hidden layers, there are significant possibilities for the number of neurons inside them, so we let Optuna automate combinations based on conditions we specified. The results can be seen in Section IV.C.1.

E. Hidden Layer Activation

ReLU has become the standard for hidden layer activation due to the function's simplicity and ability to commonly outperform other activations like Sigmoid and Tanh on image classification benchmarks. Despite its many strengths, new activation functions have been proposed to improve upon ReLU, including Google's Swish and Misra's Mish. The base model originally used Mish as the hidden layer activation. Mish has properties that would theoretically make it more ideal than alternative activation functions, like allowing for negative gradients, having a continuous first derivative, and being unbounded above and bounded below [19]. For the original base model Swish was also tried, but the model training loss exploded to infinity, so it was not used in the final network design. To determine whether to use Mish or ReLU, we tested each on the same model for 10 trials, with 25 epochs per trial, and the test results can be seen in Section IV.C.2. We chose ReLU for the hidden layer activation function as it produced the lower average validation loss.

F. Output Layer Activation

The base model used softmax activation, but we soon realized that this was not the most appropriate choice. Softmax is used often in multi-class classification problems because it outputs a probability distribution. However, we have a regression problem, so we instead used a linear activation for the output layer.

G. Optimizer

Initially, the Adam optimizer was used for training the regression-based neural network (RNN). Adam improves upon stochastic gradient descent through use of a dynamic learning rate, which allows for more effective convergence and less chance of being stuck in local minima. The reliability of Adam makes it a solid choice for many practical deep learning problems. Kingma and Ba showed that Adam outperformed four other optimizers on MNIST and CIFAR-10, two benchmark image classification datasets [20]. Although we have a regression problem and not a classification problem, Adam still makes a good starting point given that it is known to work well on RNNs.

We let Optuna do a grid search on eight different optimizers (SGD, RMSprop, Adam, Adadelta, Adagrad, Adamax, Nadam, Ftrl) built into Keras to see if there was one that might outperform Adam. Each optimizer was tested on the same model for 25 epochs. Then, the top three optimizers (Adam, Adamax, Nadam) were tested further with 50 and 100 epochs averaged over three trials. The outcome of the optimizers test can be observed in Section IV.C.3.

H. Learning Rate

The default learning rate (LR) for Adamax is 0.002, as suggested by Kingma and Ba. However, the majority of Keras's built-in optimizers have a default of 0.001. To decide the LR, we let Optuna perform a grid search in the range from 0.0001 to 0.002 with step size of 0.0001. The top 10 LRs for 10 epochs were further tested for 25 epochs. Then, the top 4 from the 25 epochs were compared for another 100 epochs. The results from the test performed on LR can be seen in Section IV.C.4.

I. Number of Epochs

Throughout our testing, we gradually increased the number of epochs in training as we found more promising configurations. Later on, we employed Keras's early stopping feature, as it helped us to thoroughly train the model while helping to prevent overfitting. Early stopping takes a parameter called patience, which stops model training after a certain number of epochs if the validation loss does not improve. Patience is commonly set somewhere between 1 and 100, so in our case we arbitrarily chose 30. It is possible that there is a better patience value for our network, however more testing for a better patience value will be left to future work. The results for testing different options for the number of epochs can be seen in Section IV.C.5.

J. Loss Function

Part of the hyperparameter optimization process led us to look at different loss functions as a possibility for improving the predictions provided by our model. We performed several studies which focused on the use of the Mean Absolute Percent Error (MAPE) loss function. MAPE produced good results for our predictions except in the data range close to zero. Hyperparameter optimization led to the examination of the Huber and LogCosh loss functions. The results of test performed while searching for an optimal loss function can be observed in Section IV.C.6.

IV. Results

In this section we show the results for the base model and discuss methods for combating overfitting. We used hyperparameter optimization to improve the base model's performance by tuning options for the number of hidden layers and neurons per layer, hidden layer activation, optimizer, learning rate, number of epochs, and loss function. We show the test results from this hyperparameter tuning. Then, we present the steps we took to reach our final model and provide a comparison of the results of the base model and final model. Finally, we introduce ensemble methods created using models produced during hyperparameter optimization, describe the tests we performed using ensemble methods, and show the results from these tests.

A. Base Model

Figure 2 shows the percent error of the base model's predictions of drag coefficient on 500 data points. Points are colored based on the angle of attack of the airfoil. The general expected error is about an order of magnitude above the actual value. There were also a significant number of points that were between one to two orders of magnitude off from the actual value.

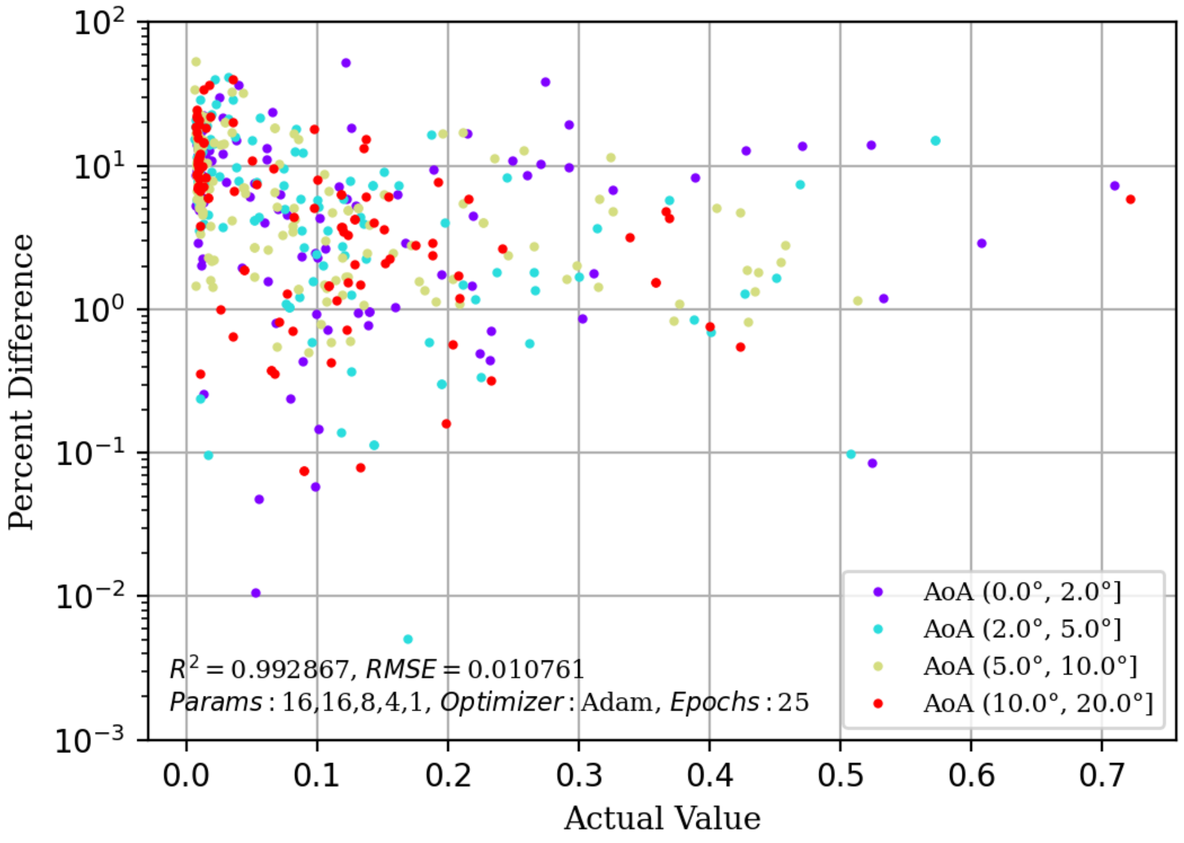


Fig. 2 Drag Original 25a

In the next sections, we compare the initial design and each of its modifications that led to the architecture of the final ML model. The choices to follow were informed by tests and visualizations generated in Optuna and TensorBoard.

B. Overfitting

One popular method to prevent overfitting is to introduce a dropout layer. Dropout randomly turns off neurons inside a neural network to avoid unintentional dependencies between neurons [21]. However, we found that dropout did not work well on our ML model. Instead, we used TensorBoard, a visualization tool integrated within TensorFlow, to verify that new models we found were not overfitting [22]. For example, in Figure 3 we show a cartoon of a model with loss curves that exhibit overfitting behavior. In this example the divergence of the training and testing loss curves at about 100 epochs indicate that training of this model should be cut off at about 100 epochs to avoid overfitting.

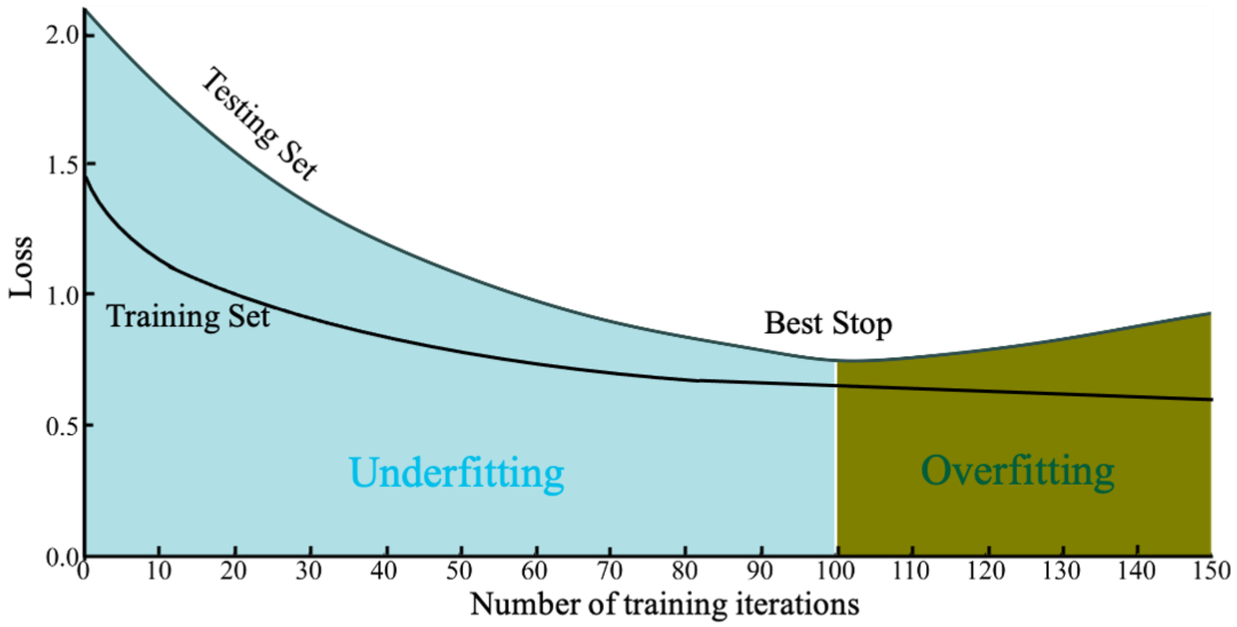


Fig. 3 Overfitting Vs Underfitting

When we increased the number of epochs, number of hidden layers, or number of neurons per layer, we made sure that the model's loss curves in TensorBoard did not exhibit the behavior in the green region in Figure 3. More specifically, we were looking to identify overfitting in the case where the training data continued to improve but the results in predictions from the testing set started to become more inaccurate.

C. Neural Network Architecture Results

In this section we present the results from the hyperparameter search using Optuna and discuss the final choice for each hyperparameter.

1. Number of Hidden Layers and Neurons per Layer

In an effort to improve our overall model results, we decided to implement the use of Optuna. Optuna is a tool that allows for sweeping through parameters and looking for better network conditions. Within Optuna we set ranges for the number of hidden layers and the number of neurons in each layer. The following are the initial conditions we passed into Optuna for hyperparameter optimization:

Number of hidden layers: 3-7

Input layer size: 14-64 neurons

Hidden layers size: 0.5x-1x of the previous layer (e.g. if the input layer has 60 neurons, the next layer would have between 30 to 60 neurons)

The parallel coordinate plot in Figure 4 shows the individual trial parameters that Optuna found with the initial set of conditions. The parameters in Figure 4 can be described as follows:

Objective value	Validation loss (MAPE)
n_layers	Number of hidden layers
n_units_inputl	Number of neurons in input layer
n_units_layer_0	Number of neurons in hidden layer 1
n_units_layer_1	Number of neurons in hidden layer 2

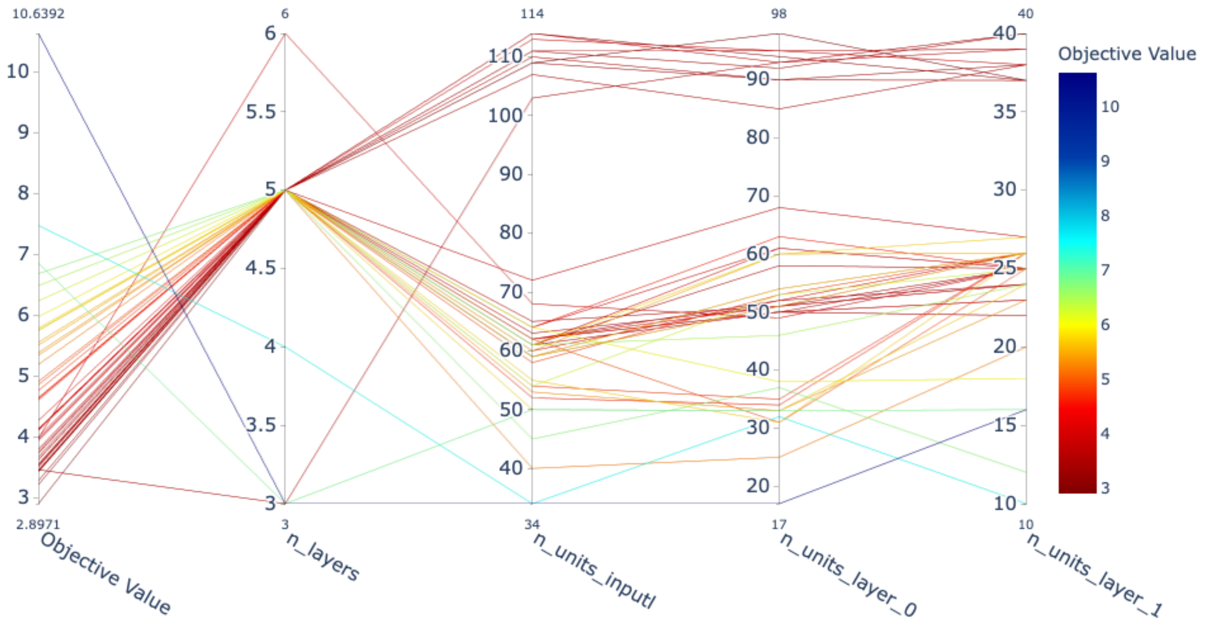


Fig. 4 Parallel Coordinate Plot for the MAPE Loss Function

In total, 760 trials were run, 709 trials were pruned, and 51 trials were completed. We see in Figure 4 that Optuna chose 5 hidden layers as the optimal number with input layer sizes in the ranges of [40, 72] and [107, 114] neurons. For the hidden layer sizes, there was no clear pattern for the first one or two hidden layers, but deeper into the network the layers tended to be about half the size of the previous layer.

After examining the results from the initial conditions, we decided to further increase the complexity of the model since it was still training very fast with 5 hidden layers. With the following conditions, Optuna was used to decide the configuration that would become the final model:

Hidden Layers	6-12
Input Layer Size	100-400 neurons
Hidden Layer Size	2-400 neurons

The parallel coordinate plot in Figure 5 shows the individual trial parameters that Optuna found with the conditions for the final model. 389 trials were run, 371 were pruned, and 18 were completed. Increasing the number of hidden layers allowed Optuna to find an input layer in the range of [329, 399] neurons, with 6 hidden layers being optimal. While 7 layer configurations were completed more often and had quite a few good objective values, it also produced several of the worst objective values. More than 7 layers appears to have diminishing returns and overfitting, with only three such completed trials. The hidden layers still decreased in size as the network got deeper, despite allowing a layer to have more neurons than the previous.

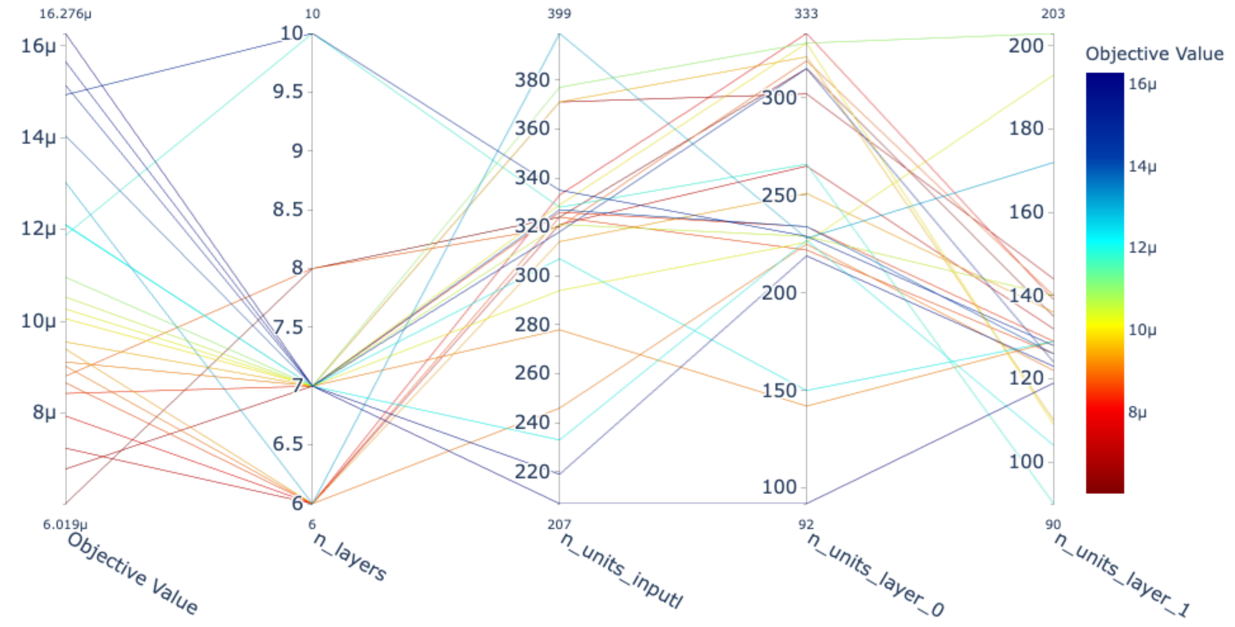


Fig. 5 Parallel Coordinate Plot for conditions that led to finding the final model

2. *Hidden Layer Activation: Mish vs. ReLU*

The results of the hyperparameter search for the hidden layer activation function are presented in Table 1, showing base model runs comparing Mish and ReLU (highlighted). The average validation loss for ReLU was 2.755%, compared to 3.097% for Mish. While 4 out of the top 10 runs were Mish, the activation also produced the bottom 5 runs, showing that Mish is less consistent than ReLU.

Rank	Activation	Val Loss (25E)	Rank	Activation	Val Loss (25E)
1	ReLU	2.4988%	11	ReLU	2.8479%
2	Mish	2.50%	12	ReLU	2.8751%
3	ReLU	2.5132%	13	Mish	2.99%
4	ReLU	2.5478%	14	ReLU	3.0354%
5	Mish	2.61%	15	ReLU	3.0434%
6	Mish	2.62%	16	Mish	3.28%
7	ReLU	2.6912%	17	Mish	3.36%
8	ReLU	2.7399%	18	Mish	3.59%
9	Mish	2.75%	19	Mish	3.62%
10	ReLU	2.7609%	20	Mish	3.65%

Table 1 Hidden Layer Activation Test

In practice, Mish outperforms ReLU in some cases on the benchmark MNIST digits dataset, but the results came from training networks with 15-25 layers and varying batch sizes. ML problems do not often need so many layers, and in our case, we did not need to split our data into batches. Mish would be more appealing had there been favorable comparisons on smaller network sizes. We decided to switch back to ReLU given its reliability and our training results shown in Table 1.

3. Optimizer: Adam vs. Nadam vs. Adamax

Table 2 shows the results of testing the same model on different optimizers, which covers all of those built into Keras. We see that Adamax, Adam, and Nadam came out on top for 25 epochs.

Optimizer	**Val Loss (25E)**	**Val Loss (50E)**	**Val Loss (100E)**	**Time (100E)**
Adamax	1.02%	0.85%	0.68%	10.3 min.
Adam	1.49%	1.11%	0.87%	10.5 min.
Nadam	1.95%	1.09%	0.82%	14.5 min.
RMSProp	2.44%	-	-	-
Adagrad	4.24%	-	-	-
Ftrl	5.53%	-	-	-
Adadelta	11.70%	-	-	-
SGD	NaN	-	-	-

Table 2 Optimizers Test

The performance of Nadam and Adamax are not surprising, as they have been closely comparable to Adam in several studies. Kandel et al. tested these three optimizers among others in histopathology image analysis and found that each excelled depending on the type of network architecture used (Adam - VGG16, DenseNet; Nadam - ResNet; Adamax - InceptionV3). However, 3 of the 4 tests did not show statistically significant results between the best and worst performance of the three optimizers, with accuracy differences of 0.15%, 0.66%, 0.81%, and 1.49%. The 1.49% difference was found on InceptionV3 network using Adamax, which may suggest its superiority in certain situations [23]. Another study conducted by Dogo et al. tested these optimizers on three randomly-selected Kaggle image datasets. Nadam came out on top for two of the three datasets, while the remaining one went to Adam. The study also compared convergence times between the optimizers, and found that Adamax converged the fastest, followed by Adam and then Nadam [24]. Ultimately, there seems to not be a clear choice between Adam, Nadam, and Adamax; it more so comes down to what works best for the problem at hand. For our study, we found that Adamax produced the best results with fast convergence (as seen in Table 2).

4. Learning Rate

Table 3 shows the results of testing different learning rates (LR) on the same model. A LR of 0.0006 produced the best result for 10, 25, and 100 epochs. For 100 epochs though, the validation losses for LRs of 0.0009, 0.0019, and 0.002 trailed close behind. Still, at 100 epochs 0.0006 had the lowest loss at 0.856%. Generally, a lower LR achieves a smoother convergence at the expense of more epochs. Here however, a LR of 0.0006 is still able to converge in a reasonable amount of time, so the time tradeoff is not as relevant as would normally be expected [25].

LR	Val Loss (10E)	Val Loss (25E)	Val Loss (100E)
0.0009	1.12%	0.98%	0.89%
0.0014	1.21%	1.24%	-
0.002	1.28%	0.92%	0.88%
0.0019	1.30%	0.96%	0.91%
0.0006	1.311%	0.847%	0.856%
0.0007	1.33%	1.12%	-
0.0012	1.34%	1.11%	-
0.0003	1.41%	-	-
0.0008	1.47%	-	-
0.0005	1.51%	-	-

Table 3 Learning Rate Test

5. Number of Epochs

Figure 6 shows the TensorBoard graph for the best model run for 1000 epochs. We see that in the first 200 epochs, the validation loss curve closely follows the training loss curve. Beyond 200 epochs, the validation curve starts to plateau and separate from the training curve, suggesting signs of overfitting. Keras's early stopping callback on average stopped the model at around 200 epochs, so that is the number we used to train the best model.

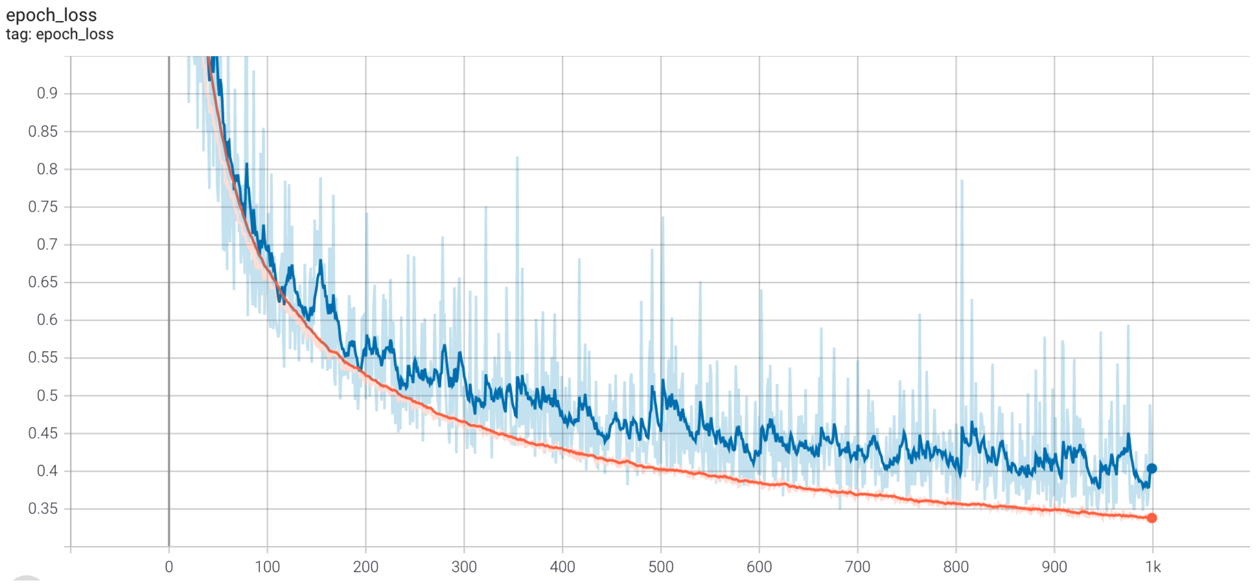


Fig. 6 Number of Epochs

6. Loss Functions

During our look at the loss function, we noticed that there was a limitation of MAPE on data points close to 0 when performing initial training of the drag model; it seemed as though the weights were failing to update properly with MAPE in this model (Figure 7). We considered Huber and Logcosh as alternative loss functions, and the result for Logcosh is shown in Figure 8. The use of Logcosh repaired the issues with training drag seen when using the MAPE function and produced more a reasonable looking plot (Figure 8). In future work, Logcosh will likely be preferred over the Huber loss function. Though they share the same advantages, the Huber loss function would require tuning an additional hyperparameter, which could be costly to determine.

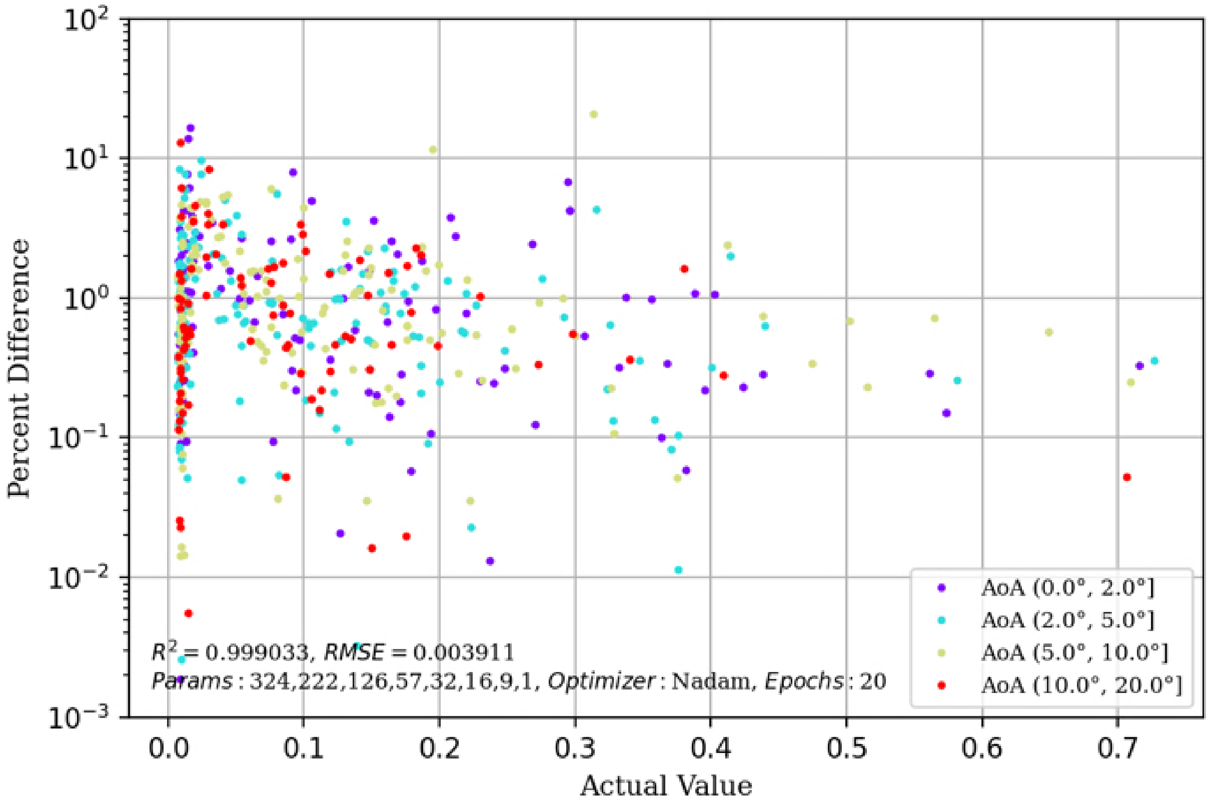


Fig. 7 MAPE Loss Function Test Results

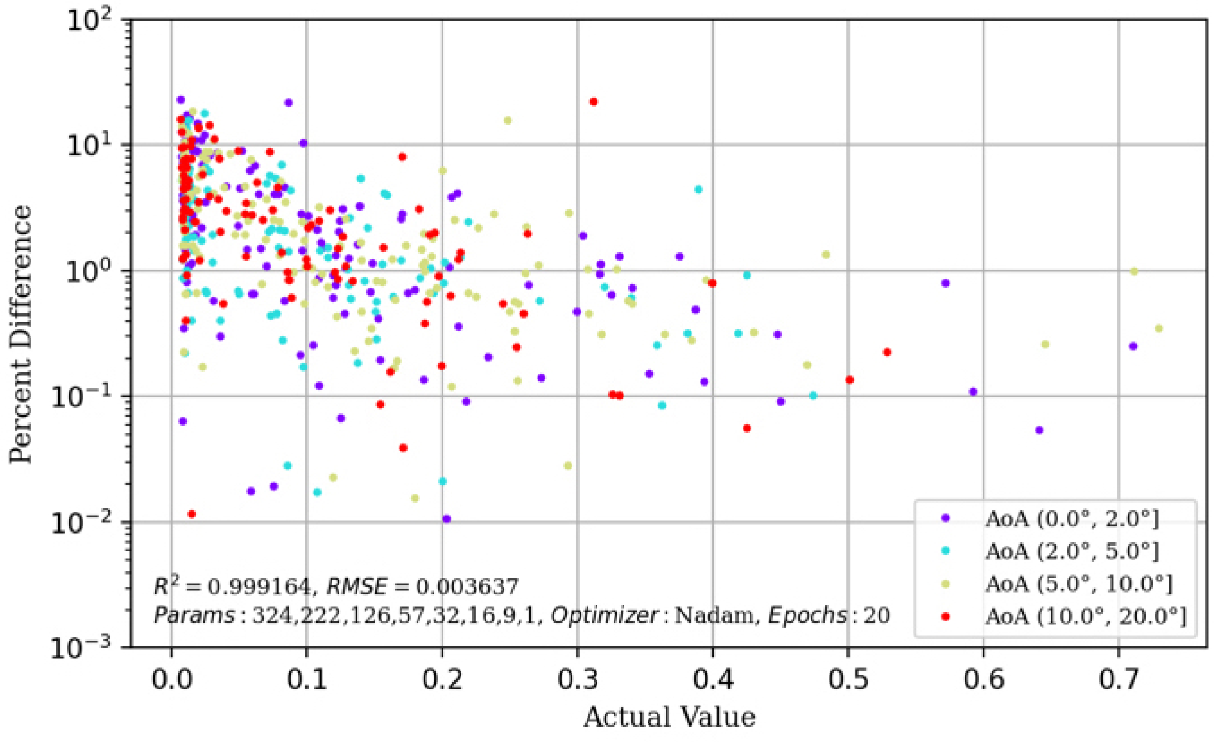


Fig. 8 Initial Logcosh Loss Function Test Results

D. Summary of Steps to Final Model

Here we will look at a summary of the steps taken to get from the starting point of linear interpolation, transitioning to building the base model using a neural network, followed by using hyperparameter optimization. We will then take a short look at the best model located through this process before ensembling methods were applied.

1. Linear Interpolation

The bar graphs in Figure 9 show the distributions of error in the predictions for points with various drag coefficients (mu) using the linear interpolation method to compute the air tables. We focus on lower values of drag coefficient because this is where we see the highest degree of error, due to there being the lowest margin for error. The error is tabulated at each normalized forward flight velocity, where mu is calculated by

$$\text{mu} = \text{vehicle flight speed}/\text{rotor tip speed}.$$

We can see that linear interpolation tends to produce errors in the range of ±250%. There are some outliers outside of ±250% with worse results such as those seen in mu = 0.24. In mu = 0.24 we can see errors in the range of -500% up to almost 750%. There are options to reduce error found in linear interpolation such as sampling more points, but using these extra samples requires additional computation time using CFD models; this work is looking to improve model accuracy without sampling more CFD points.

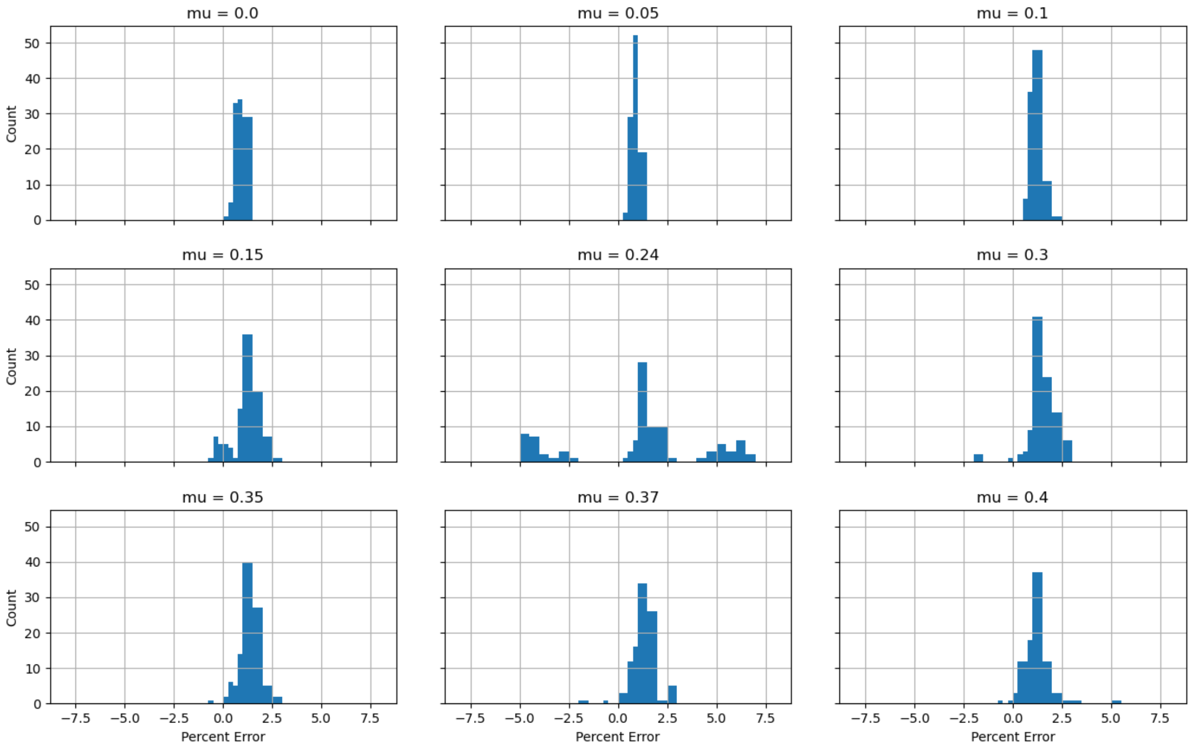


Fig. 9 Histograms of percent error using linear interpolation

Table 4 displays a summary of the statistics shown in Figure 9 for the percent errors of the drag coefficient predictions using linear interpolation. We see that the mean of percent error ranges from ~85% error to ~143% error.

mu	**0.0**	**0.05**	**0.1**	**0.15**	**0.24**	**0.3**	**0.35**	**0.37**	**0.4**
count	102	102	102	102	102	102	102	102	102
mean	0.8534	0.8605	1.1152	1.074594	1.147324	1.433962	1.272401	1.218535	1.107003
std	0.2493	0.2043	0.3042	0.722038	3.281998	0.703469	0.543612	0.68337	0.70893
min	0.2463	0.44157	0.540728	-0.577864	-4.90635	-1.81702	-0.53454	-1.96081	-0.61557
25%	0.6648	0.705023	0.885208	0.812251	0.73259	1.104198	0.965715	0.823362	0.707655
50%	0.8294	0.840901	1.045421	1.170881	1.249883	1.376993	1.240295	1.215328	1.045802
75%	1.0240	0.980409	1.326864	1.560228	2.281964	1.877479	1.654725	1.629042	1.393702
max	1.471513	1.420221	2.08377	2.501025	8.219547	2.744838	2.559548	2.948367	5.123247

Table 4 Statistics for percent errors in the prediction of data points with various drag coefficients (mu) using linear interpolation

2. Initial Neural Network for Base Model

With the use of neural networks, we can see a reduction in error in comparison to using linear interpolation, as evident by a general tightening of the spread of the error histograms (Figure 10). Though the range of error for the ML model at mu = 0.24 is still about the same as with linear interpolation, the frequency of these high error values is reduced. These results in Figure 10 show that the neural network approach is going to consistently provide estimates for the power coefficient tables that will be more accurate than the values produced using the linear interpolation approach (Figure 9).

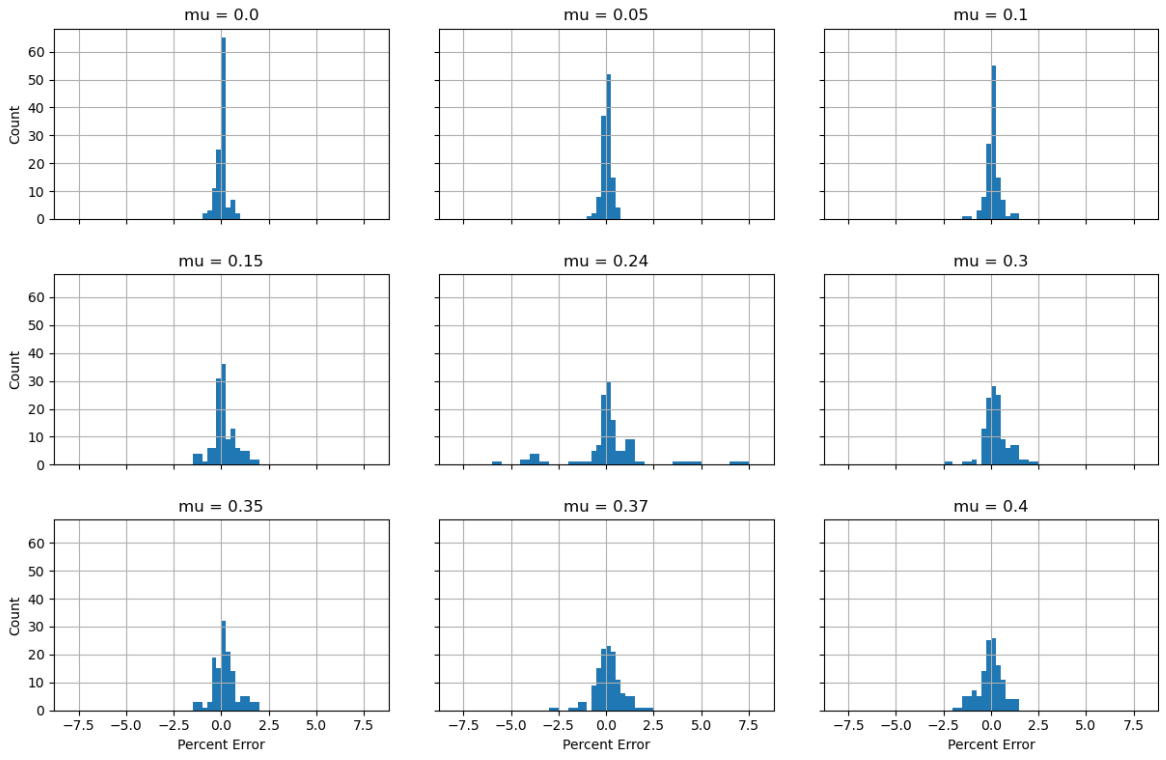


Fig. 10 Histograms of percent error using the neural network model

Table 5 displays a summary of statistics for the percent errors of lower drag coefficient (mu) predictions using neural networks. We see that the largest mean percent error is ~20% error for predictions using the neural network, which is significantly lower than the percent error of predictions using linear interpolation (Table 4). Because we are observing drag coefficients that are close to zero, small differences between the drag coefficients and the predictions result in larger percent error. Therefore, the impact of these high percentages of error is low on the accuracy of the power coefficient that results from the performance tables that are generated by the network.

mu	**0.0**	**0.05**	**0.1**	**0.15**	**0.24**	**0.3**	**0.35**	**0.37**	**0.4**
count	119	119	119	119	119	119	119	119	119
mean	0.019481	0.045969	0.094344	0.127665	0.084728	0.205783	0.155414	0.092191	0.067463
std	0.279356	0.234753	0.31285	0.517913	1.641484	0.567834	0.526382	0.634942	0.971601
min	-0.89689	-0.98214	-1.11886	-1.31219	-5.93081	-2.12056	-1.22168	-2.76316	-1.92399
25%	-0.05203	-0.06545	-0.05823	-0.10428	-0.1226	-0.08305	-0.19554	-0.22951	-0.27194
50%	0.035129	0.058321	0.071778	0.063685	0.06694	0.201968	0.133247	0.079503	0.016649
75%	0.083077	0.177541	0.219738	0.438618	0.426983	0.422871	0.399985	0.379661	0.287101
max	0.770263	0.611293	1.090276	1.656562	7.114196	2.373645	1.715405	2.491554	8.722475

Table 5 Statistics for percent errors in the prediction of data points with various drag coefficients (mu) using a neural network

3. Hyperparameter Optimization Spreadsheet of Modifications

The graphs in Figure 11 compile all the results from parameter sweeps performed by Optuna, from the base model and all the intermediate models produced before arriving at the optimal RNN. The network iteration refers to the subsequent model configurations beginning from the base model to the final model. The top plot shows the percentage of data points under selected thresholds of error. The results of the final model before we tried ensembling methods are significantly better in all percent error thresholds (i.e. 20%, 10%, 5%, 2.5%, 1%, 0.5%). In the bottom plot we see an improvement in both the RMSE and MAPE from the base model to the final model.

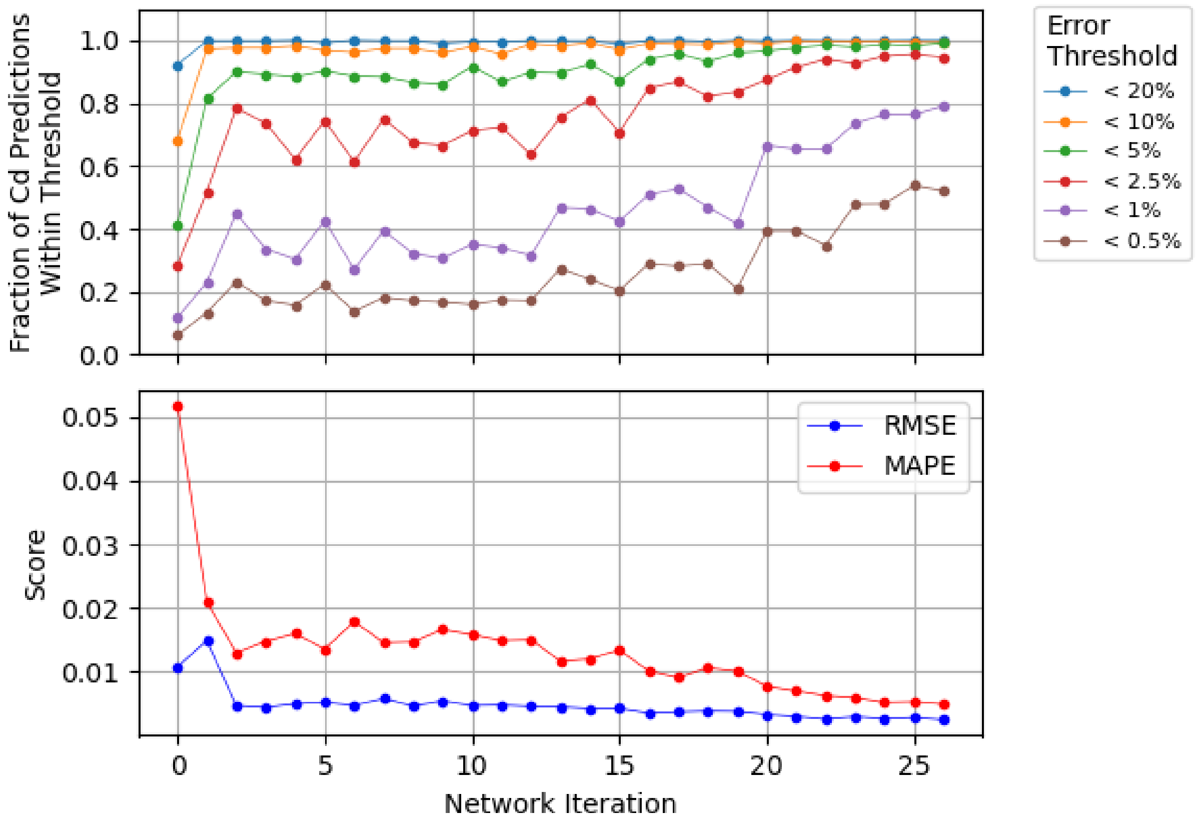


Fig. 11 Full Model Results

Figure 12 shows a summary of all the intermediate modifications we made to the base model to arrive at the final model. Five-hundred points were randomly sampled for clarity of visualization for each graph. We can see in the plot labeled "Base Model" in Figure 12 that the base model yielded errors well 10 percent across all angles of attack (AoA). After examining the results from the base model, the initial modification included increasing the number of hidden layers from 3 to 4, doubling the size of the input layer from 16 to 32 neurons, and using a linear activation in the output layer in place of the softmax activation. After the first modification the model performed better with very few points having error over 10 percent (plot labeled "Modification 1" in Figure 12). The second modification included the introduction of hyperparameter optimization with Optuna. We began tuning the number of neurons in the input layer by searching in a range of [14, 64] neurons, setting up the criteria to reduce each proceeding layer by 0.5-1 times the size of the previous layer. After the second modification, we can see that the model performed even better, with most of the points being below 10 percent error (plot labeled "Modification 2" in Figure 12). The error range for points more closely fits around no error rather than clustering on either side of 10 percent error. We then made the decision to use the ReLU activation in the hidden layers instead of Mish. We increased the number of hidden layers from 4 to 5 and searched the range of [32, 128] for the number of neurons in the input layer, which we labeled the third modification; with this model the eror is clustering closer to 1 percent (plot labeled "Modification 3" in Figure 12). To further improve model performance, we decided to continue tuning the hyperparameters in Optuna. Based on the Optuna results, we increased the number of hidden layers from 5 to 6, decreased the learning rate from 0.001 to 0.0006, modified the number of neurons in the input layer to the range [128, 400], used the Nadam optimizer in place of Adam, and let early stopping determine the number of epochs. After this fourth modification, we see that the the expected error continues to trend lower (plot labeled "Modification 4" in Figure 12). Finally, through hyperparameter optimization, we made the choice to use the Adamax optimizer instead of Nadam, resulting in the final model. In the final model (plot labeled "Final Model" in Figure 12), we see that model is showing the lowest expected margin of error.

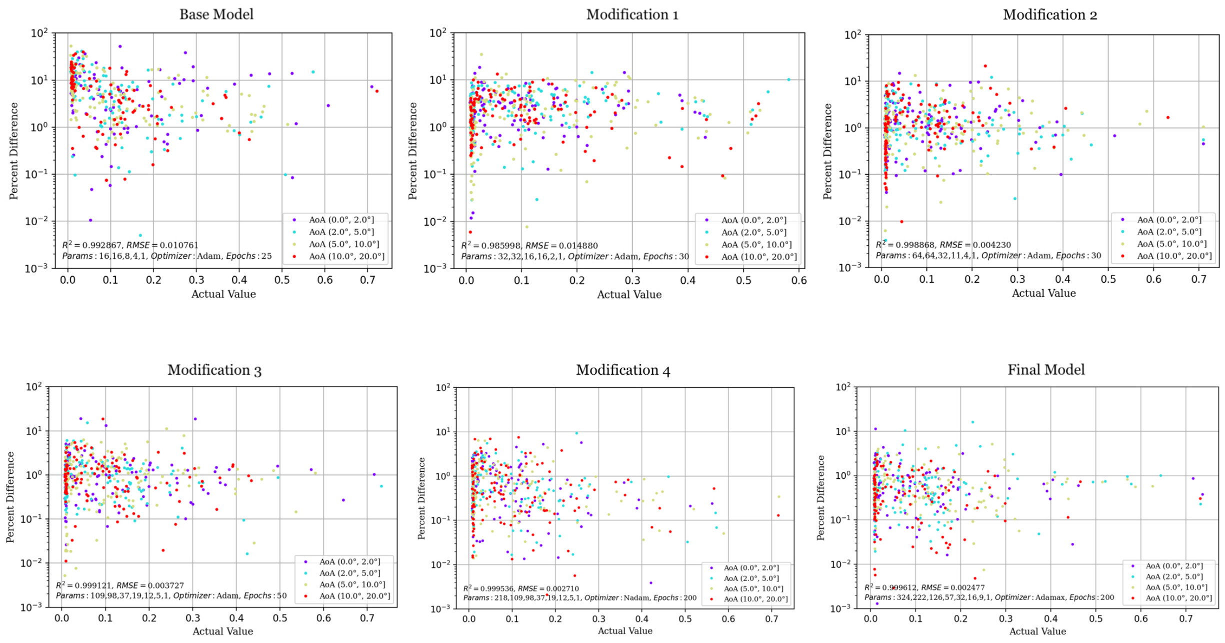


Fig. 12 Full Test Results with Modifications

4. Final Model Before Ensemble Method

Table 6 shows the transition from the base model to the final model, specifically focusing on the choices for the hyperparameters of each model. The choices for the final model were determined through hyperparameter optimization by Optuna. The reasoning behind the adjustment of each hyperparameter from the initial choice for the base model to the final choice has been discussed in Section IV.C.

Summary	**(Base → Final)**
Hidden layer activation:	Mish → ReLU
Optimizer:	Adam → Adamax
Learning rate:	0.001 → 0.0006
Number of hidden layers:	3 → 6
Number of neurons per layer:	[16, 16, 8, 4, 1] → [324, 222, 126, 57, 32, 16, 9, 1]
Epochs:	25 → 200

Table 6 Model Configuration Parameters: Base Model to Final Model

Table 7 shows a distribution comparing the percent difference performance of the final model against the base model. The "<20%" column indicates the percentage of data points predicted with under a 20% error, the "<10%" column indicates the percentage of data points predicted with under a 10% error, and so on. The final model shows significant improvement over the base model in accuracy of points with a 100% likelihood that a point will never be beyond 20% error. The final model also has the ability to predict points within 5% error 99.2% of the time which is a significant improvement over the base model only being accurate within 5% error 41.4% of the time. Table 7 shows that the implementation of hyperparameter optimization improved model performance.

Model	**<20%**	**<10%**	**<5%**	**<2.5%**	**<1%**	**<0.5%**
Base	92.20%	68.20%	41.40%	28.40%	11.80%	6.20%
Final	100.00%	99.40%	99.20%	94.60%	79.00%	52.20%

Table 7 Percent Difference Thresholds (Base vs. Final)

E. Error Propagation Analysis

For this section, we analyzed how the errors of the neural network (NN) predictions propagate downstream to errors in rotor performance metrics. We prepared equivalent linear interpolation and NN models by training/fitting to identical airfoil coefficient data. The NN used the optimal architecture found by Optuna prior to ensembling. Using CFD and uniform random sampling from the design space, we generated a new set of test data consisting of 150 C81 tables. The test set was not seen by either model prior to evaluation. We then used each model to generate predicted tables corresponding to each "ground-truth" table in the test set. The result is three sets of 150 tables from three different sources. A summary of the drag coefficient prediction errors (absolute-percent-error) from each model is shown in Table 8. Overall, the statistics indicate that the NN model performs slightly better than linear interpolation.

	mean	**std dev**	**median**
Drag Coeff. (% error) Interpolation	2.80	7.6	0.80
Drag Coeff. (% error) Neural Network	2.37	3.7	1.16

Table 8 Interpolation vs. Neural Network Error Results

Next, we used the three sets of tables as inputs to the rotorcraft comprehensive code, RCAS, to make predictions of the rotor performance. The tables are used to describe a modified airfoil applied to the tip of the main rotor blades (0.84R to 1.0R). We then conducted a speed sweep by incrementing the forward flight speed from $\mu = 0.0$ (hover) to $\mu = 0.4$. For this analysis, we use the rotor power coefficient, C_p, as the performance metric because it is sensitive to the airfoil drag.

Figure 13 shows the predicted C_p values versus actual C_p values for both linear interpolation (left) and the NN model (right). In both methods, we removed points for simulations that did not converge in RCAS. Both models performed well, on average, as can be seen by the ±2.5% margin included in the plots, where 2.5% was chosen arbitrarily. We observed that RCAS with the interpolation-generated tables has more outliers and tends to systematically over-predict the rotor power. The exact cause of this is being investigated. We also observe that RCAS with the NN-generated tables does not suffer from the same phenomenon and yields a more Gaussian error distribution.

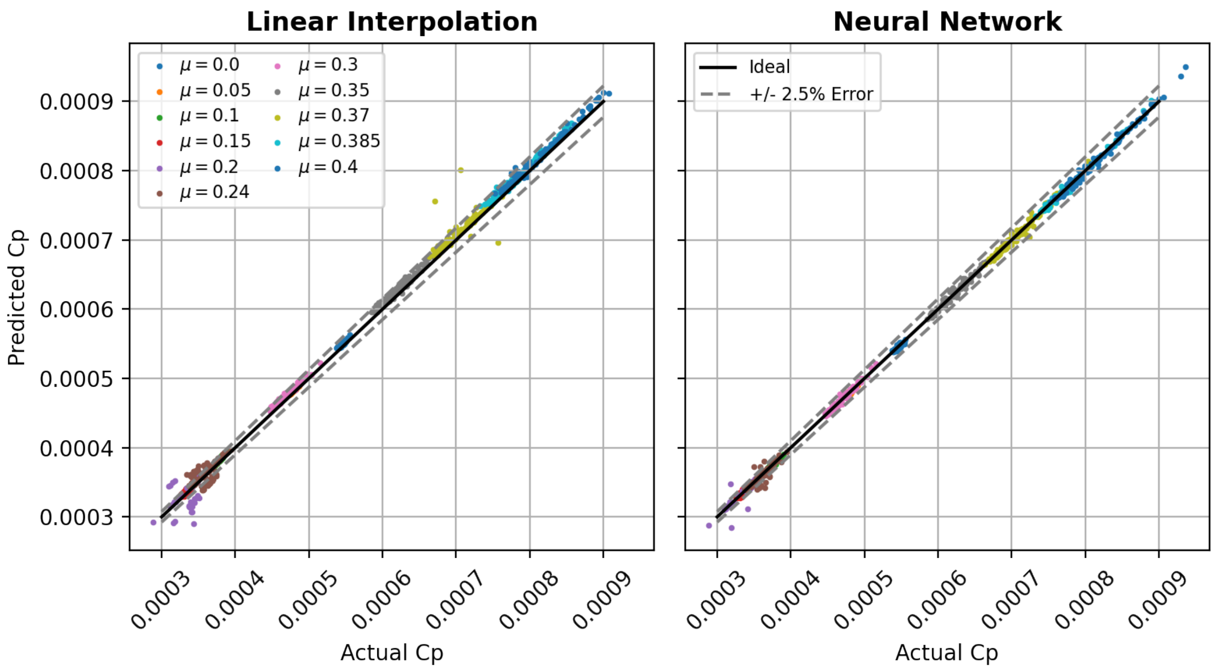


Fig. 13 Rotor power coefficient predicted vs. actual values for linear interpolation (left) and neural network model (right)

Figure 14 shows predicted L/D_E values versus actual L/D_E values for both linear interpolation (left) and the NN model (right). The result is interesting because the same network architecture was used to predict both the airfoil lift and drag coefficients. The results are encouraging considering that the network was designed without consideration for the lift coefficient (though the same procedure can be readily applied).

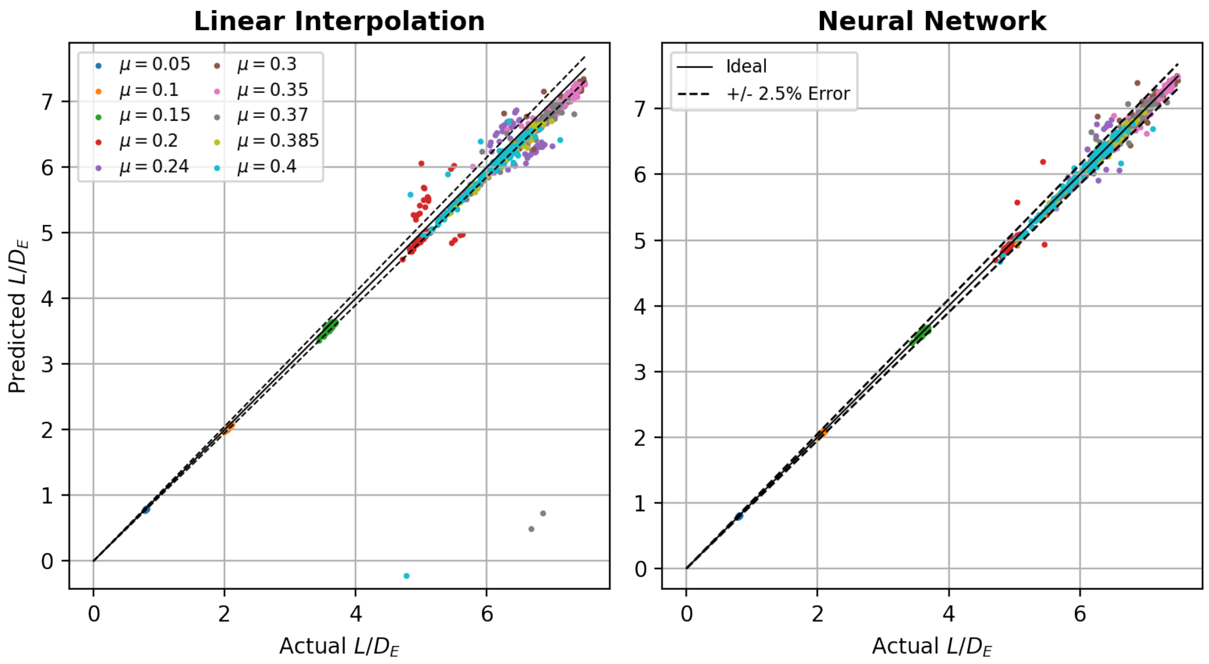


Fig. 14 L/D_E predicted vs. actual values for linear interpolation (left) and neural network model (right)

F. Ensemble Methods

After the initial success of introducing a neural network into our problem and implementing hyperparameter optimization, we decided to explore additional methods to further improve model performance of the final model discussed in the previous section. We considered ensemble methods. Ensembling is an optimization method often used in top-performing models for machine learning problems [26]. The idea behind ensembling is to combine the predictions of multiple models to leverage the differing strengths of each model for more robust predictions. Ensembling can take the form of a simple statistical analysis, such as taking the average or median of predictions. Alternatively, it can involve stacking, where multiple models are concatenated to form a new model that is designed to learn how to best combine the stacked predictions. There are several different kinds of ensembling methods, and we are going to briefly introduce a few of them in this section.

Concatenation is the process of combining models side-by-side, which can suffer from an explosion of dimensionality as a result. The simple average (or mean) method is performed by taking the average predictions of each model. Unlike concatenation, simple average does not suffer from increased dimensionality problems; it makes the assumption that the data is normally-distributed. The weighted average method, which is another option, takes tensor outputs and multiplies the outputs by weights, then linearly combines the weights. The total of the weights multiplied by all models must add to one for the weighted average method to work. The weighted average method allows the network's designer to designate the contribution of each model to the final prediction by altering the weights.

In the following sections we discuss the results of applying neural network ensembling methods (Section IV.F.1), such as the simple average and weighted average, using the best models from Optuna. We will follow the Optuna results with a look at gradient boosting ensembling methods (Section IV.F.2) using a well known library called XGBoost.

1. Neural Network Ensembles

IV.F.1.1 Simple Average Figure 15 is the plot of the best neural network found by Optuna before we considered any ensemble methods. The simplest ensemble method is to take the average of multiple neural network predictions. Figure 16 shows the improvement using three models chosen for the ensemble (model paprameters on the right side of the figure). The three chosen models are within the top five models found in Optuna.

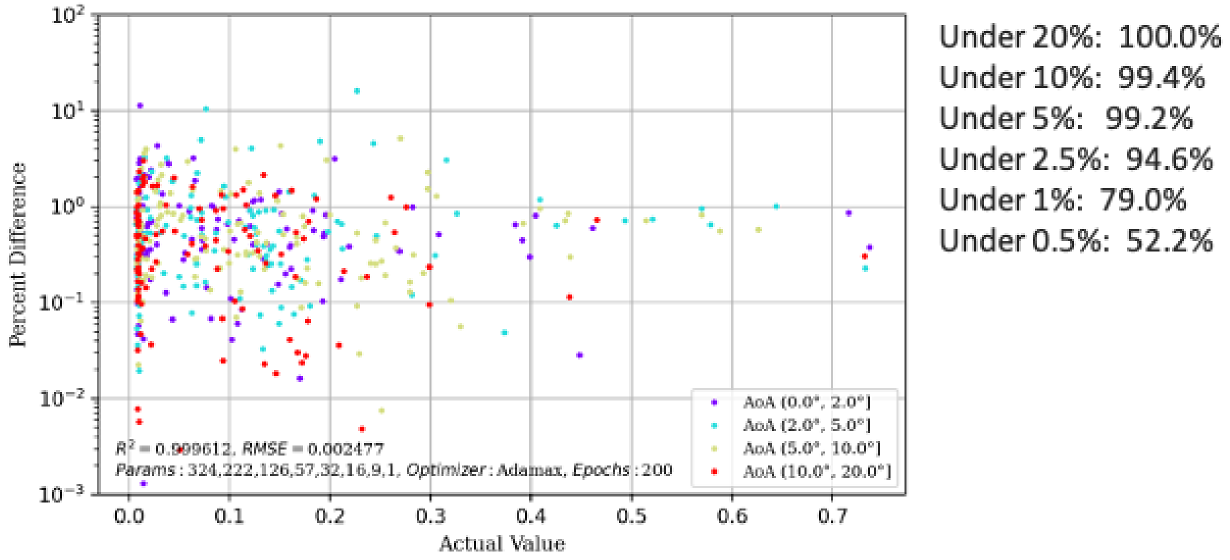


Fig. 15 Sample output from best neural network before ensembling

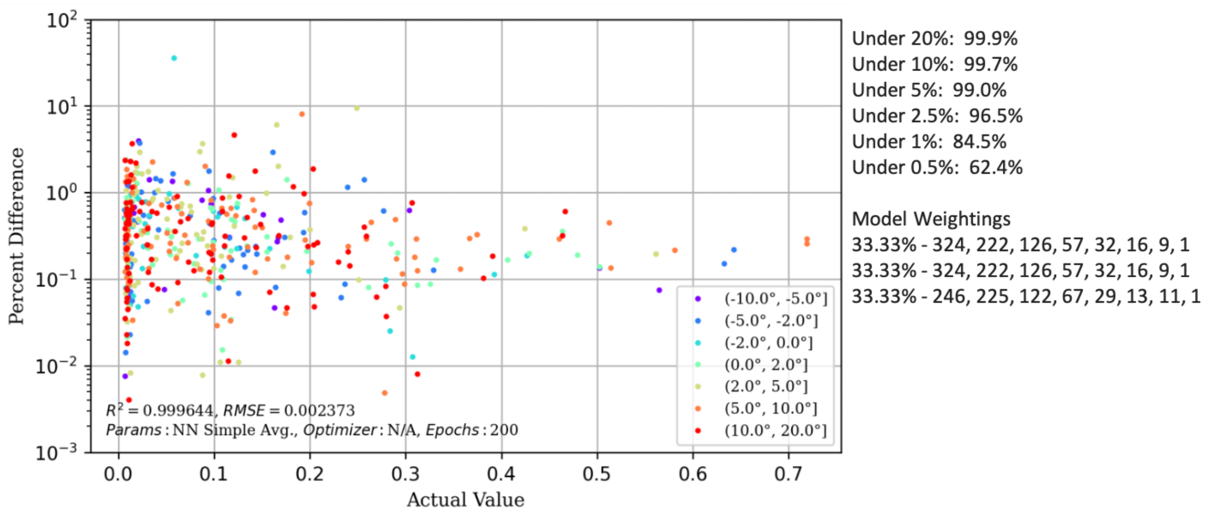


Fig. 16 Neural Network with Simple Average Ensemble Method

In Figure 16 we already see a noticeable improvement over the single neural network model in Figure 15, with higher R^2, lower RMSE, and a higher percentage of points predicted under 2.5%, 1%, and 0.5% error.

IV.F.1.2 Weighted Average A weighted average allows us to place more or less weight on the predictions of certain neural networks. A Cartesian product grid search was conducted to test possible weight vectors, and the optimal combination ended up being 50%, 21.43%, 28.57%. However, compared to the simple average plot and error percentages in Figure 16, there was almost no improvement using the weighted average (Figure 17).

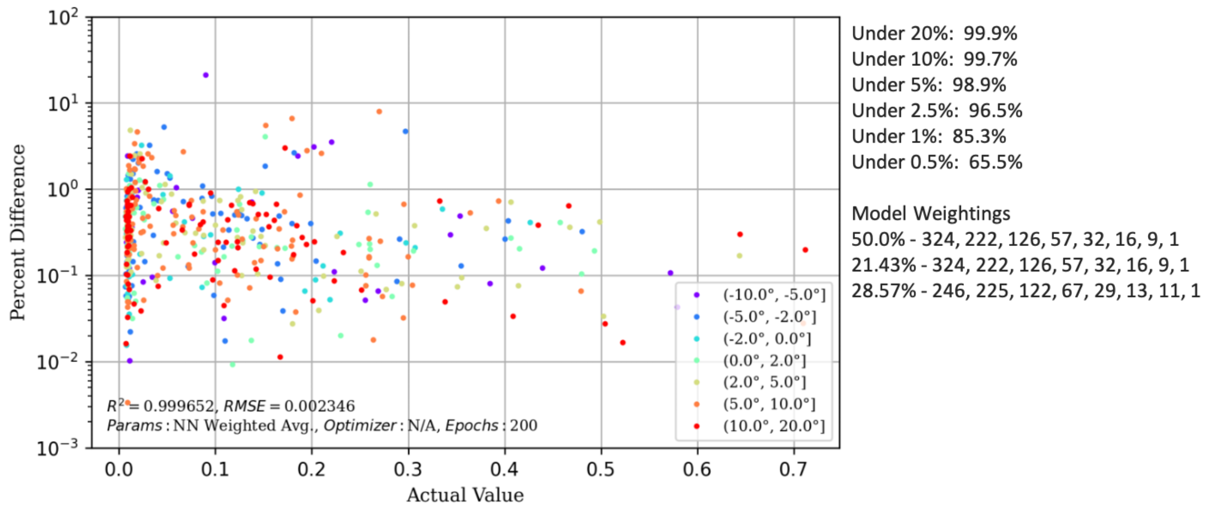


Fig. 17 Neural Network with Weighted Average Ensemble Method

2. *Gradient Boosting Ensembles*

While applying ensembling methods to the network models created by Optuna, Gradient Boosting Ensembling also is considered. Gradient boosting is a more classical machine learning technique using decision tree methods, and has been known to work well for most problems [27, 28]. It is a type of ensembling as each iteration a new decision tree learns to correct mistakes from previous decision trees [29]. Its advantage over a neural network is that it has a simpler structure and is much faster to train, which makes this method a choice when the dataset is not large (as in our case). The most popular gradient boosting libraries are XGBoost, CatBoost, and LightGBM. For this work, we use XGBoost based on comparison of results using the Python library known as AutoML [30]. Even though XGBoost is the slowest of the three gradient boosting libraries mentioned here, the difference in training speed is negligible for our problem.

IV.F.2.1 XGBoost Initial Model The default XGBoost model was initially run on our dataset with the parameters defined in the table below. Max Depth is the maximum depth of a tree. Increasing this value makes the model more complex and more prone to overfitting. N_Estimators is the number of gradient boosted trees. Learning Rate is the boosting learning rate. Max Bin is the maximum number of discrete bins to bucket continuous features. Subsample is the subsample ratio of the training instances. Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. Lambda is the L2 regularization term on weights [31].

Max Depth	6
N_Estimators	100
Learning Rate	0.3
Max Bin	256
Subsample	1
Lambda	0

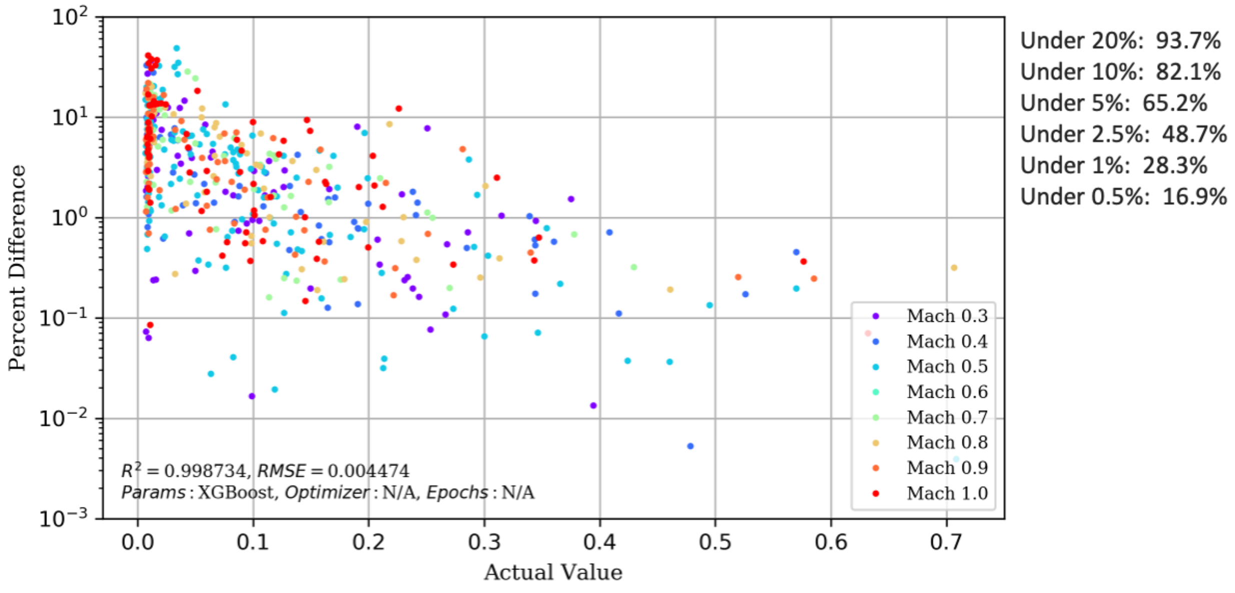


Fig. 18 XGBoost Base Model

The base model only took a few seconds to train, but we observe in Figure 18 that the performance is unsatisfactory, as there are a large cluster of points above 10% error. We needed to tune the model to see how promising it was for locating an improvement over the best neural network model before applying ensembling.

IV.F.2.2 Single Best XGBoost Optuna is used for automating the optimization process of these hyperparameters modified by XGBoost. These initial conditions were specified for Optuna to conduct a hyperparameter search:

Max Depth	5 to 20
Learning Rate	0.01 to 0.3, step 0.01
N_Estimators	1000, with early_stopping_rounds of 10 iterations
Subsample	0.4 to 1.0, step 0.05
Alpha, Lambda, Gamma	0 to 5

Alpha is the L1 regularization term on weights, and gamma is the minimum loss reduction required to make a further partition on a leaf node of the tree. The larger gamma is, the more conservative the algorithm will be [31]. From the slice plot of the optimization trials, all the best models had an alpha and gamma value of 0, so these two parameters were eliminated from the search. In addition, all the top models had a subsample rate of at least 0.6, so the range was reduced accordingly:

Max Depth	5 to 20
Learning Rate	0.01 to 0.3, step 0.01
N_Estimators	1000, with early_stopping_rounds of 10 iterations
Subsample	0.6 to 1.0, step 0.05
Lambda	0 to 5

After tuning our model with Optuna, we see an order of magnitude improvement in the results in Figure 19 compared to the XGBoost base model results in Figure 18. Most points are now under 10% error, and its error ranges are on-par with some deeper neural networks. Its <20%, <10%, and <5% error percentages are worse than several neural networks. However, its <1% error percentage is similar to the fourth best NN, and the <0.5% error percentage is close to the third best NN. While considerably more complex than the base model, this tuned model took less than two minutes to train.

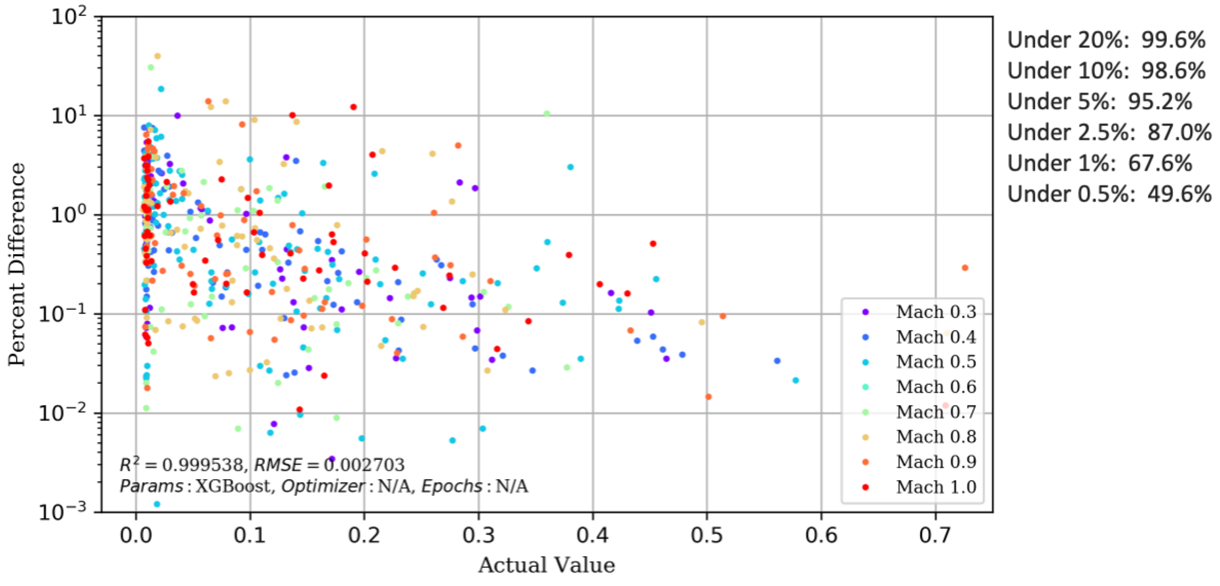


Fig. 19 Single Best XGBoost Model Results

IV.F.2.3 Voting Regressor (Top 3) The simplest ensemble model for XGBoost is using's scikit-learn's VotingRegressor, which takes several base estimators and then simply averages the individual predictions to form a final prediction. For the base regressors, we simply took the top three XGBoost models from Optuna's hyperparameter tuning. Compared to the best individual XGBoost model in Figure 19, we observe another noticeable improvement in the results using VotingRegressor (Figure 20), with higher percentages for the error ranges <5%, <2.5%, <1%, and <0.5%.

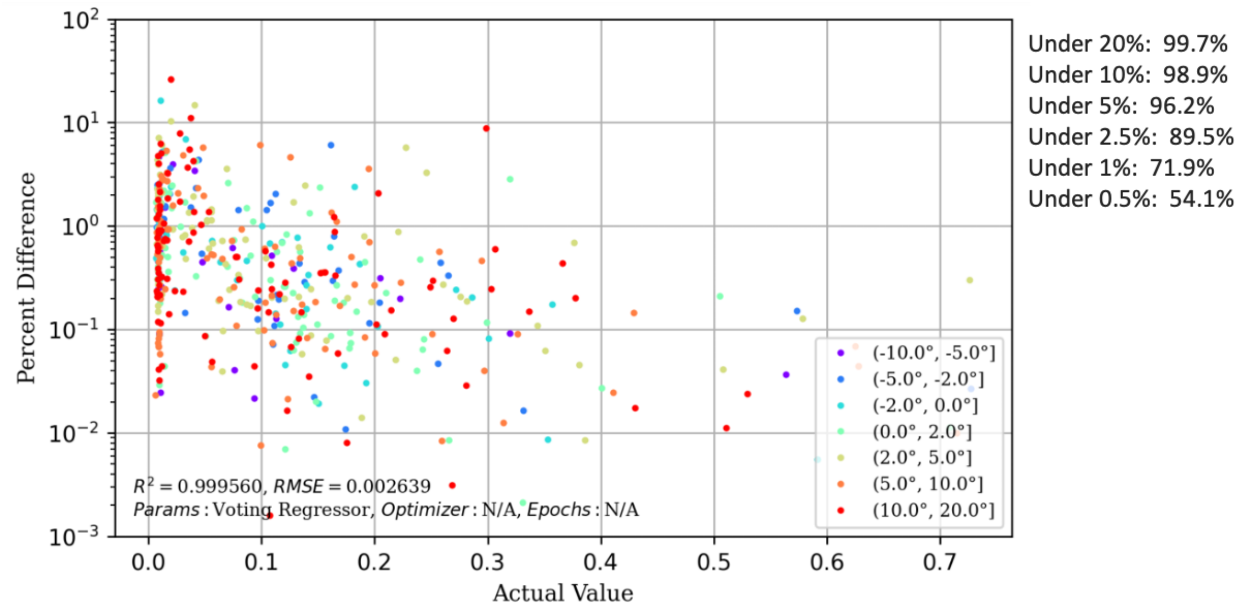


Fig. 20 VotingRegressor Results

IV.F.2.4 Weighted Average VotingRegressor provides an additional parameter called 'weights' should we want to take a weighted average of the XGBoost predictions. However, the best weights found from grid search produced no improvement. The plot is not shown here, but the results from this test are included in Figure 24.

3. Hybrid Ensembles

It is recommended that the VotingRegressor take conceptually different machine learning models as base regressors. In Figure 20 we were using three XGBoost models, which are conceptually very similar. Scikit-learn does not provide an easy way to combine neural networks and XGBoost. However, as a baseline, we can do similar statistical analysis like we did for neural networks alone. Figures 21 and 22 show the results for the average and median of predictions from the top 5 neural networks (Table 9) and top 3 XGBoost models (Table 10) from Optuna, resulting in a hybrid ensemble model:

Parameters	Optimizer
324, 222, 126, 57, 32, 16, 9, 1	Adamax
324, 222, 126, 57, 32, 16, 9, 1	Nadam
321, 265, 132, 53, 21, 15, 6, 1	Adamax
246, 225, 122, 67, 29, 13, 11, 1	Adamax
246, 225, 122, 67, 29, 13, 11, 1	Nadam

Table 9 Top 5 Neural Network Model Configurations (LR = 0.0006, 200 epochs each)

Max Depth	Learning Rate	Bagging Fraction	Random State
20	0.0138	0.867	2020
19	0.017	0.867	48
20	0.015	0.86	2020

Table 10 Top 3 XGBoost Model Configurations (n_estimators = 750)

While the average has its uses, taking the median may be more robust, since it is not highly affected by outliers from skewing results. In the rare event that one of the models fails to train, the result will also not be ruined. In both cases, the ensemble outperforms both neural network and XGBoost-only ensembles, as we see another large improvement in percentages for < 2.5%, < 1%, and < 0.5% error. The median (Figure 22) does slightly better than a simple average (Figure 21). So, the overall best model is the median hybrid ensemble.

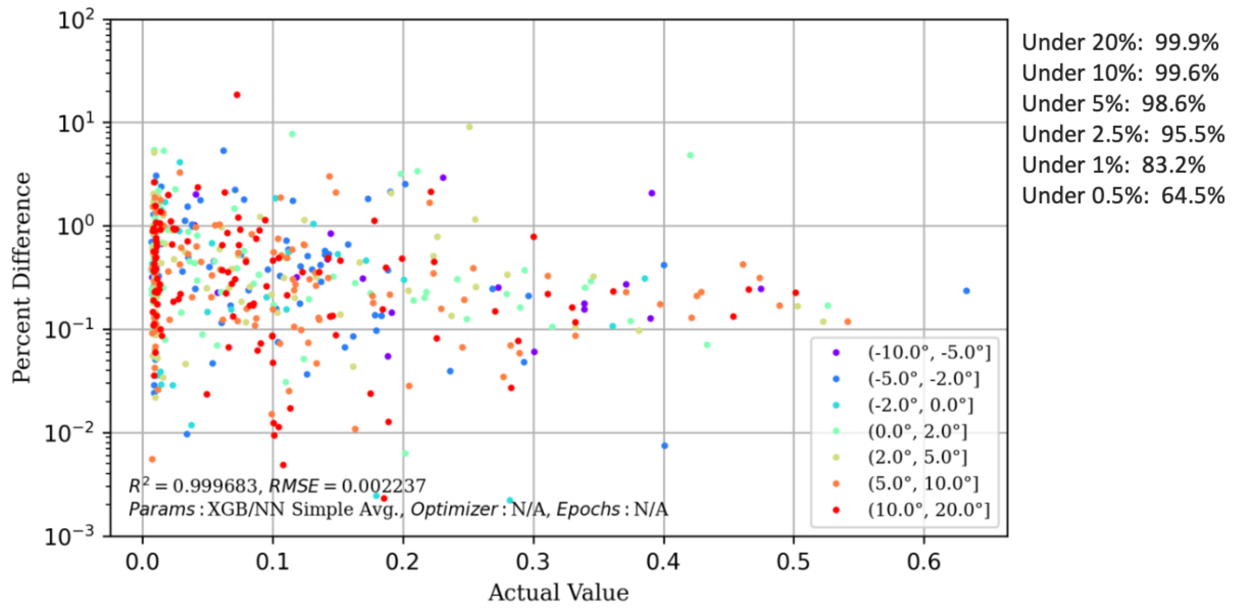


Fig. 21 Neural Network + XGBoost Hybrid Average Ensemble Results

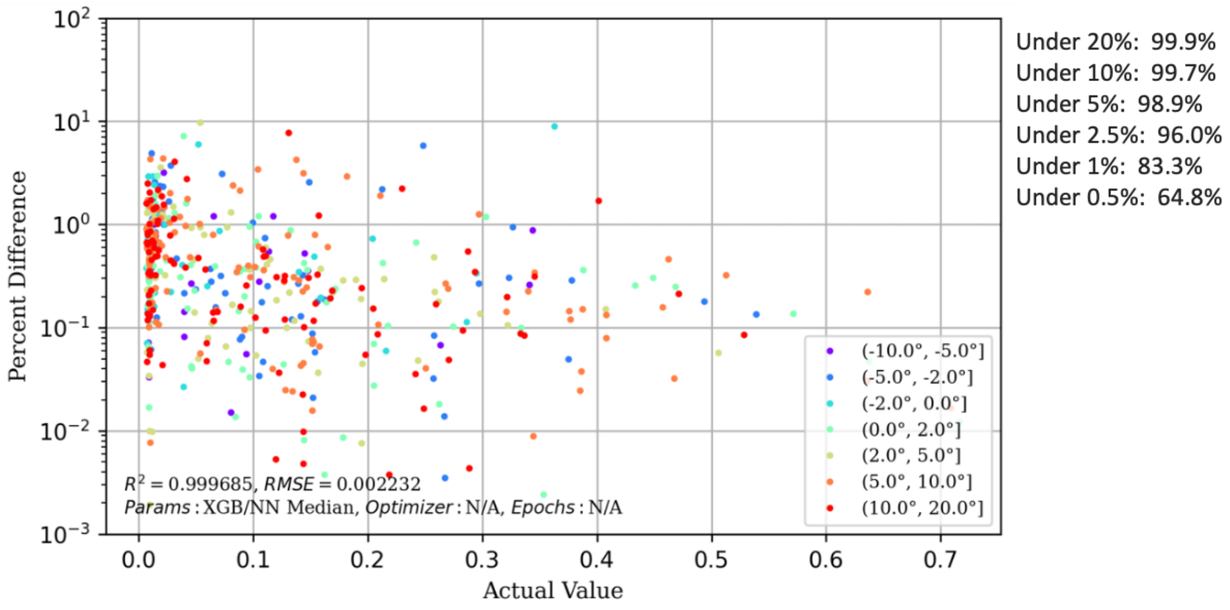
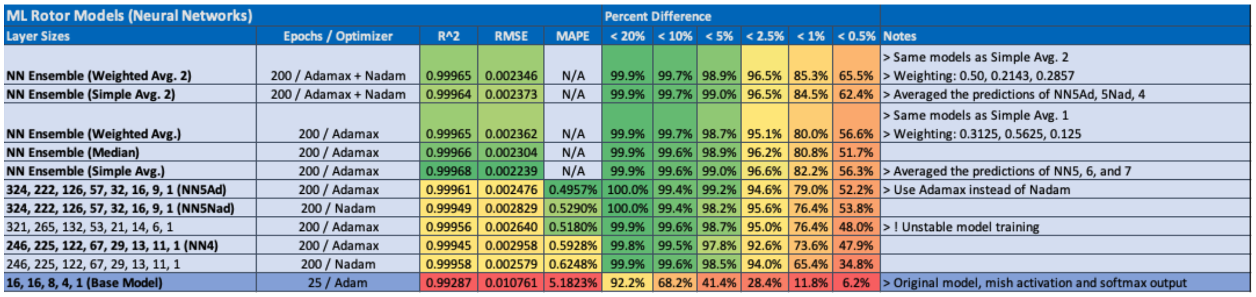


Fig. 22 Neural Network + XGBoost Hybrid Median Ensemble Results

G. Summary of Optuna Hyperparameter Optimization, XGBoost, and Ensembling Methods

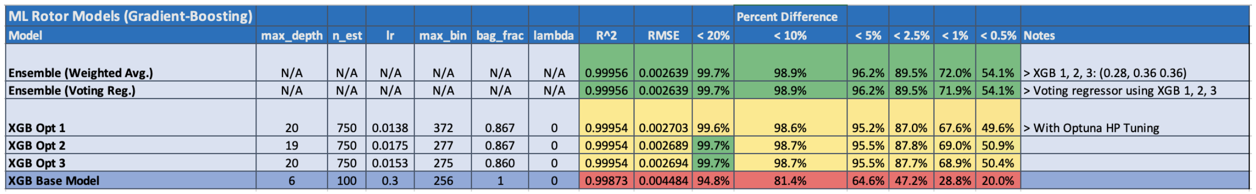
We close the results section with a description and summary tables showing the best results from neural network modeling, XGBoost, and ensembling methods. The first table we are going to look at is the neural network results (Figure 23). We can see a high degree of improvement as we moved from the base model, transitioning to models created by hyperparamter optimization, and lastly to ensemble models created by applying simple average, weighted average, and median ensembling methods leveraging the best models located by Optuna. The results of this table show that as we move from the base hand-generated model to applying hyperparamter optimization and then to ensembling methods, we see significant improvements in the number of points within each error range. A notable observation is that the best model created by ensembling with neural networks was able to generate 96.5% of the points within 2.5% error.



ML Rotor Models (Neural Networks)					Percent Difference						
Layer Sizes	Epochs / Optimizer	R^2	RMSE	MAPE	< 20%	< 10%	< 5%	< 2.5%	< 1%	< 0.5%	Notes
NN Ensemble (Weighted Avg. 2)	200 / Adamax + Nadam	0.99965	0.002346	N/A	99.9%	99.7%	98.9%	96.5%	85.3%	65.5%	> Same models as Simple Avg. 2 > Weighting: 0.50, 0.2143, 0.2857
NN Ensemble (Simple Avg. 2)	200 / Adamax + Nadam	0.99964	0.002373	N/A	99.9%	99.7%	99.0%	96.5%	84.5%	62.4%	> Averaged the predictions of NN5Ad, 5Nad, 4
NN Ensemble (Weighted Avg.)	200 / Adamax	0.99965	0.002362	N/A	99.9%	99.7%	98.7%	95.1%	80.0%	56.6%	> Same models as Simple Avg. 1 > Weighting: 0.3125, 0.5625, 0.125
NN Ensemble (Median)	200 / Adamax	0.99966	0.002304	N/A	99.9%	99.6%	98.9%	96.2%	80.8%	51.7%	
NN Ensemble (Simple Avg.)	200 / Adamax	0.99968	0.002239	N/A	99.9%	99.6%	99.0%	96.6%	82.2%	56.3%	> Averaged the predictions of NN5, 6, and 7
324, 222, 126, 57, 32, 16, 9, 1 (NN5Ad)	200 / Adamax	0.99961	0.002476	0.4957%	100.0%	99.4%	99.2%	94.6%	79.0%	52.2%	> Use Adamax instead of Nadam
324, 222, 126, 57, 32, 16, 9, 1 (NN5Nad)	200 / Nadam	0.99949	0.002829	0.5290%	100.0%	99.4%	98.2%	95.6%	76.4%	53.8%	
321, 265, 132, 53, 21, 14, 6, 1	200 / Adamax	0.99956	0.002640	0.5180%	99.9%	99.6%	98.7%	95.0%	76.4%	48.0%	> ! Unstable model training
246, 225, 122, 67, 29, 13, 11, 1 (NN4)	200 / Adamax	0.99945	0.002958	0.5928%	99.8%	99.5%	97.8%	92.6%	73.6%	47.9%	
246, 225, 122, 67, 29, 13, 11, 1	200 / Nadam	0.99958	0.002579	0.6248%	99.9%	99.6%	98.5%	94.0%	65.4%	34.8%	
16, 16, 8, 4, 1 (Base Model)	25 / Adam	0.99287	0.010761	5.1823%	92.2%	68.2%	41.4%	28.4%	11.8%	6.2%	> Original model, mish activation and softmax output

Fig. 23 Neural Network Summary of Results

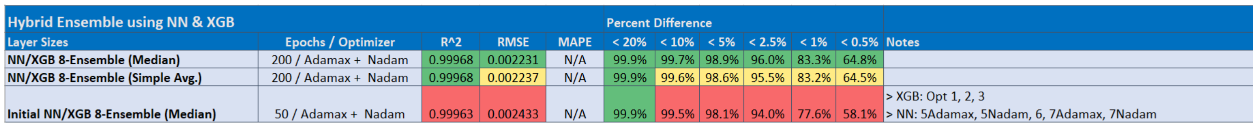
The results of working with XGBoost is seen in Figure 24 found below. The initial XGBoost model performance is viewable on the last line of the figure. After the intial model was created, the team followed the same approach we had done when designing the neural networks by applying Optuna to make architectural configuration choices. Optuna produced several XGBoost models with increasingly accurate results. We can observe that Optuna started its approach by increasing the depth from 6 to 13 and eventually settled around 19 to 20 as the ideal depth for the XGBoost network. We can also see an immediate improvement over the base XGBoost model to the models using Optuna with a 5% increase in the number of points within 20% error. There was also a significant increase in the number of points within 10%, 5%, 2.5%, 1%, and 0.5% error. Results of the points within the targeted error ranges continued to marginally increase throughout each Optuna iteration featured in the table. It should be noted that XGBoost did not produce a better model every iteration, but tried many different combinations of hyperparameters to find the networks which identified points within smaller regions of error. After applying Optuna to the XGBoost model, we had the result of 96.2% of the points within 5% error, but we were still looking to improve the model. We then applied the VotingRegressor and weighted average ensemble methods (described in Sections IV.F.2.3 and IV.F.2.4 respectively) using the best models produced from XGBoost with Optuna runs.



ML Rotor Models (Gradient-Boosting)										Percent Difference					
Model	max_depth	n_est	lr	max_bin	bag_frac	lambda	R^2	RMSE	< 20%	< 10%	< 5%	< 2.5%	< 1%	< 0.5%	Notes
Ensemble (Weighted Avg.)	N/A	N/A	N/A	N/A	N/A	N/A	0.99956	0.002639	99.7%	98.9%	96.2%	89.5%	72.0%	54.1%	> XGB 1, 2, 3: (0.28, 0.36 0.36)
Ensemble (Voting Reg.)	N/A	N/A	N/A	N/A	N/A	N/A	0.99956	0.002639	99.7%	98.9%	96.2%	89.5%	71.9%	54.1%	> Voting regressor using XGB 1, 2, 3
XGB Opt 1	20	750	0.0138	372	0.867	0	0.99954	0.002703	99.6%	98.6%	95.2%	87.0%	67.6%	49.6%	> With Optuna HP Tuning
XGB Opt 2	19	750	0.0175	277	0.867	0	0.99954	0.002689	99.7%	98.7%	95.5%	87.8%	69.0%	50.9%	
XGB Opt 3	20	750	0.0153	275	0.860	0	0.99954	0.002694	99.7%	98.7%	95.5%	87.7%	68.9%	50.4%	
XGB Base Model	6	100	0.3	256	1	0	0.99873	0.004484	94.8%	81.4%	64.6%	47.2%	28.8%	20.0%	

Fig. 24 Gradient-Boosting Summary of Results

The final figure discussed here is Figure 25 which shows what we are calling the hybrid ensemble model. The hybrid ensemble model is a combination of the best neural networks and XGBoost models prior to ensembling. It was interesting to combine the benefits of neural networks and decision tree support to get a better result than either individual method was able to accomplish.



Hybrid Ensemble using NN & XGB					Percent Difference						
Layer Sizes	Epochs / Optimizer	R^2	RMSE	MAPE	< 20%	< 10%	< 5%	< 2.5%	< 1%	< 0.5%	Notes
NN/XGB 8-Ensemble (Median)	200 / Adamax + Nadam	0.99968	0.002231	N/A	99.9%	99.7%	98.9%	96.0%	83.3%	64.8%	
NN/XGB 8-Ensemble (Simple Avg.)	200 / Adamax + Nadam	0.99968	0.002237	N/A	99.9%	99.6%	98.6%	95.5%	83.2%	64.5%	
Initial NN/XGB 8-Ensemble (Median)	50 / Adamax + Nadam	0.99963	0.002433	N/A	99.9%	99.5%	98.1%	94.0%	77.6%	58.1%	> XGB: Opt 1, 2, 3 > NN: 5Adamax, 5Nadam, 6, 7Adamax, 7Nadam

Fig. 25 Hybrid Ensembles Summary of Results

V. Conclusion

Due to the computational time requirements of running CFD models for estimating the rotorcraft airfoil performance being expensive, having a ML surrogate model would allow for estimating a large number of potential airfoil designs in a much shorter time-frame. In this work we provided an analysis of the effects of applying hyperparameter optimization to the base ML surrogate model and employing ensembling to improve accuracy, seeking an alternative for estimating the airfoil tables that were previously produced by running CFD simulations.

In a prior work use of linear interpolation between CFD points was used to estimate drag coefficient values for an airfoil. Though linear interpolation proved to be computationally efficient and reasonably accurate, the benefits of this approach begin to decrease on large, irregularly sampled parameter spaces. Also, as part of that prior work a RNN was manually designed to address the difficulties of applying linear interpolation to an irregularly sampled parameter space; this initial model proved to be insufficient, in part, because it suffered from overfitting. In this work we investigate approaches to improve on the base RNN model that was manually formulated; we decided to explore hyperparameter optimization using the Optuna library. The addition of hyperparameter tuning to find the optimal values for our neural network architecture resulted in a significant improvement in model performance over the base model. We then performed an error propagation study to determine how the errors of the neural network (NN) predictions propagate downstream to errors in rotor performance metrics. A notable observation was that linear interpolation suffered from the issue of systematically over-predicting the rotor power, but the neural network did not experience the same phenomenon. Next, we began exploring ensemble methods to build on the success of hyperparameter optimization, leading to an analysis of neural network and gradient boosting ensembling methods. We performed tests using the simple average and weighted average ensembling schemes for our top-performing neural networks. Though both schemes provided similar results, we did observe a noticeable improvement in model performance over our Optuna-designed neural network. Continuing our exploration of ensemble methods we employed XGBoost for ensembling using decision trees. We manually designed an XGBoost model that yielded unsatisfactory results, so we modified the hyperparameters in our XGBoost model using Optuna, which improved performance. We used the VotingRegressor and weighted average ensembling schemes on our top 3 Optuna-designed XGBoost models, both tests producing similar results but still an improvement over the best XGBoost model chosen by Optuna. Finally, we combined the top 5 neural networks and top 3 XGBoost models designed using Optuna into a hybrid ensemble model, using the simple average and median ensembling schemes. The resulting models from both of these tests proved to perform better than all of our other models, with the median hybrid ensemble model being determined to be the best performing model due to its robustness.

The results of this study yielded a ML model for estimating the drag coefficient of an aircraft airfoil that was able to decrease analysis time and improve accuracy. Future work for this effort includes applying this machine learning workflow to the lift and moment coefficients that will allow computation of full air tables (CL, CD, and CM) that can be used in vehicle performance calculations. In addition to a more robust method for generating models, we also seek to increase the number of ensembling methods to find an accurate model for predicting drag, lift, and moment coefficients.

References

[1] Mahgerefteh, H., Atti, O., and Denton, G., “An interpolation technique for rapid CFD simulation of turbulent two-phase flows,” *Process Safety and Environmental Protection*, Vol. 85, No. 1, 2007, pp. 45–50.

[2] Palmer, G., “Construction of CFD solutions using interpolation rather than computation with the ADSI code,” *47th AIAA aerospace sciences meeting including the new horizons forum and aerospace exposition*, 2009, p. 141.

[3] Amsallem, D., *Interpolation on manifolds of CFD-based fluid and finite element-based structural reduced-order models for on-line aeroelastic predictions*, Stanford University, 2010.

[4] Vinuesa, R., Azizpour, H., Leite, I., Balaam, M., Dignum, V., Domisch, S., Felländer, A., Langhans, S. D., Tegmark, M., and Fuso Nerini, F., “The role of artificial intelligence in achieving the Sustainable Development Goals,” *Nature communications*, Vol. 11, No. 1, 2020, pp. 1–10.

[5] Recht, B., “A tour of reinforcement learning: The view from continuous control,” *Annual Review of Control, Robotics, and Autonomous Systems*, Vol. 2, 2019, pp. 253–279.

[6] Vinuesa, R., and Brunton, S. L., “Enhancing computational fluid dynamics with machine learning,” *Nature Computational Science*, Vol. 2, No. 6, 2022, pp. 358–366.

[7] Hammond, J., Pepper, N., Montomoli, F., and Michelassi, V., “Machine Learning Methods in CFD for Turbomachinery: A Review,” *International Journal of Turbomachinery, Propulsion and Power*, Vol. 7, No. 2, 2022, p. 16.

[8] Wang, B., and Wang, J., “Application of artificial intelligence in computational fluid dynamics,” *Industrial & Engineering Chemistry Research*, Vol. 60, No. 7, 2021, pp. 2772–2790.

[9] Gatski, T. B., and Speziale, C. G., “On explicit algebraic stress models for complex turbulent flows,” *Journal of fluid Mechanics*, Vol. 254, 1993, pp. 59–78.

[10] Hardesty, L., “Explained: neural networks,” *MIT News*, Vol. 14, 2017.

[11] Brahme, A., *Comprehensive biomedical physics*, Newnes, 2014.

[12] Wang, L., Shao, C., Wang, H., and Wu, H., “Radial basis function neural networks-based modeling of the membrane separation process: hydrogen recovery from refinery gases,” *Journal of Natural Gas Chemistry*, Vol. 15, No. 3, 2006, pp. 230–234.

[13] Elsayed, K., and Lacor, C., “Modeling, analysis and optimization of aircyclones using artificial neural network, response surface methodology and CFD simulation approaches,” *Powder technology*, Vol. 212, No. 1, 2011, pp. 115–133.

[14] Allen, L. D., Lim, J. W., Haehnel, R. B., and Dettwiller, I. D., “Rotor Blade Design Framework for Airfoil Shape Optimization with Performance Considerations,” *AIAA Scitech 2021 Forum*, 2021, pp. 1–21. https://doi.org/10.2514/6.2021-0068, URL https://arc.aiaa.org/doi/abs/10.2514/6.2021-0068.

[15] “Keras: the Python deep learning API,” , 2022. URL https://keras.io/.

[16] "Optuna: A hyperparameter optimization framework — Optuna 3.0.0 documentation," , 2022. URL https://optuna.readthedocs.io/en/stable/index.html.

[17] Team, K., "Keras documentation: KerasTuner API," , 2022. URL https://keras.io/api/keras_tuner/.

[18] Uzair, M., and Jamil, N., "Effects of Hidden Layers on the Efficiency of Neural networks," *2020 IEEE 23rd International Multitopic Conference (INMIC)*, 2020, pp. 1–6. https://doi.org/10.1109/INMIC50486.2020.9318195.

[19] Misra, D., "Mish: A Self Regularized Non-Monotonic Activation Function," , 2019. https://doi.org/10.48550/ARXIV.1908.08681, URL https://arxiv.org/abs/1908.08681.

[20] Kingma, D. P., and Ba, J., "Adam: A Method for Stochastic Optimization," , 2014. https://doi.org/10.48550/ARXIV.1412.6980, URL https://arxiv.org/abs/1412.6980.

[21] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R., "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, Vol. 15, No. 56, 2014, pp. 1929–1958. URL http://jmlr.org/papers/v15/srivastava14a.html.

[22] "TensorBoard," , 2022. URL https://www.tensorflow.org/tensorboard.

[23] Kandel, I., Castelli, M., and Popovič, A., "Comparative Study of First Order Optimizers for Image Classification Using Convolutional Neural Networks on Histopathology Images," *Journal of Imaging*, Vol. 6, No. 9, 2020. https://doi.org/10.3390/jimaging6090092, URL https://www.mdpi.com/2313-433X/6/9/92.

[24] Dogo, E. M., Afolabi, O. J., Nwulu, N. I., Twala, B., and Aigbavboa, C. O., "A Comparative Analysis of Gradient Descent-Based Optimization Algorithms on Convolutional Neural Networks," *2018 International Conference on Computational Techniques, Electronics and Mechanical Systems (CTEMS)*, 2018, pp. 92–99. https://doi.org/10.1109/CTEMS.2018.8769211.

[25] Zulkifli, H., "Understanding Learning Rates and How It Improves Performance in Deep Learning," , Jan. 2018. URL https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10.

[26] Demir, N., "Ensemble methods: Elegant techniques to produce improved machine learning results," *Toptal Engineering Blog*, 2016.

[27] Chen, T., and Guestrin, C., "Xgboost: A scalable tree boosting system," *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[28] Hancock, J. T., and Khoshgoftaar, T. M., "CatBoost for big data: an interdisciplinary review," *Journal of big data*, Vol. 7, No. 1, 2020, pp. 1–45.

[29] Aliyev, V., "Gradient Boosting Classification explained through Python," , Oct. 2020. URL https://towardsdatascience.com/gradient-boosting-classification-explained-through-python-60cc980eeb3d.

[30] He, X., Zhao, K., and Chu, X., "AutoML: A survey of the state-of-the-art," *Knowledge-Based Systems*, Vol. 212, 2021, p. 106622.

[31] Developers, X., "XGBoost Parameters," , 2022. URL https://xgboost.readthedocs.io/en/stable/parameter.html.