Deep Learning Neural Network Based Convergence Criteria for Computational Fluid Dynamics

a project presented to

The Faculty of the Department of Aerospace Engineering

San José State University

in partial fulfillment of the requirements for the degree

Master of Science in Aerospace Engineering

by

Joshua F. Diaz

December 2022

D. Dalle, PhD[1], P. Papadopoulos, PhD[2]
Industry Advisor [1], Faculty Advisor [2]

ABSTRACT

Determining if a computational fluid dynamics (CFD) solution has converged is critical to the quality of the solution. Assuming the grid and solver settings for a CFD simulation are adequate then the CFD simulation will conclude when a solution satisfies specified convergence criteria. This paper presents a comparison between convergence classification based on simple convergence criteria and a supervised machine learning model. The machine learning model utilizes the open-source artificial intelligence (AI) framework TensorFlow paired with the hyperparameter optimizer KerasTuner. The CFD simulations were conducted using the NASA developed viscous flow solver, FUN3D. The training and validation dataset consist of 300 cases covering an ascent trajectory of a near axisymmetric rocket. Each case is classified as either converged or not converged by an expert user based on the iterative history of three force coefficients and one moment coefficient. A comparison between the simple algorithms and the training data shows monitoring physical quantities of interest for asymptotic behavior, both with and without feature smoothing, are not able to achieve satisfactory accuracy; 67.3% and 71.3% respectively for a threshold of ± 0.0005 over 500 iterations. Two datasets were created to test each method's performance: an ascent trajectory comprised of 25 cases, and 50 high supersonic cases respectively. For the high supersonic dataset, each model achieved on average within 1% of the user's results. For the ascent trajectory dataset, a machine learning model which sampled 2000 iterations achieved on average between 1.82-2.94%; a machine learning model which sampled 1000 iterations achieved between 2.21-4.57%; the asymptotic criteria achieved on average between 3.26-4.31%; and the asymptotic with smoothing criteria achieved on average between 3.25-4.33%. For each test dataset, a substantial decrease in computational resources was observed relative to the user.

Acknowledgements

First, I would like to thank the Dr. Dalle for his initial insight of the application of machine learning for convergence criteria, as well as his guidance and support of the project. I am also grateful to my entire team at NASA Ames Research Center Computational Aerosciences Branch for their contributions to the project and their patience throughout the completion of my degree.

Second, I would like to thank my faculty advisor Dr. Papadopoulos for his support in both my career development and project. His experience and guidance have been invaluable for opening doors and creating the opportunity to pursue this project.

Last, but not least, I want to thank my fiancé for her tremendous patience and support over the past two years. You have truly helped make a difficult journey feel easy.

Contents

List of Figures

List of Tables

Symbols

Symbol	**Definition**	**Units (SI)**
CA	Axial Force Coefficient	-------
CY	Side Force Coefficient	-------
CN	Normal Force Coefficient	-------
CLM	Pitching Moment Coefficient	-------
Acronyms		
AI	Artificial Intelligence	-------
API	Applications Program Interface	-------
CFD	Computational Fluid Dynamics	-------
CLV	Common Launch Vehicle	-------
CNN	Convolutional Neural Network	-------
ML	Machine Learning	-------
MNIST	Modified National Institute of Standards and Technology	-------
RANS	Reynold's Averaged Navier-Stokes	-------
ReLu	Rectified Linear Unit	-------
RMS	Root-Mean-Squared	-------
SLS	Space Launch System	-------

1. Introduction

As most undergraduate engineers taking their first fluid mechanics course learn, the Navier-Stokes equations are non-linear and through analytical restrictions, an iterative approach can be utilized to achieve a solution. As they continue their education and begin to resolve full flow fields using computational fluid dynamics (CFD) simulations, one of the first questions asked is, "How do we know when it is done?". They are usually provided an answer about the residuals dropping a few magnitudes and leveling out[1]. Although looking to observe asymptotic behavior in either the properties of interest or residuals can be a good guide, simulation outputs may never exhibit this ideal behavior. Noise or oscillations in the solutions may result from poor grid quality, solver settings, or unsteady flow phenomena[2]. When an ideal output cannot be achieved, it is often left to "engineering judgement" to determine if a case has sufficiently converged on a solution.

For large scale databases, leaving each CFD case to be judged by a user can not only be time consuming, but introduces inconsistency given differences in each contributing user's convergence criteria. Looking to resolve these hindrances is NASA Ames Research Center's Computational Aerosciences Branch. The branch is responsible for creating large scale databases for the three versions of the Artemis Program's Space Launch System (SLS)[3]. Each database consists of tens of thousands of CFD cases covering either ascent or booster separation. Currently, each case is judged on two pages of outputs. The first page consists of approximately nine plots depicting force and moment coefficients and residuals. The second page consist of flow visualization for various views. Each time a case completes its specified iteration count, a case report is compiled and judged for convergence. If the case is deemed not yet converged, either the solver settings are altered, or additional iterations are specified.

To reduce the manual oversight of this workflow, automated convergence criteria can be implemented. In the past, simple algorithms have been insufficient for the many flow solvers, such as Cart3d and FUN3D. Machine learning models offer a more complex approach through pattern recognition. The machine learning models presented in this paper seek to classify each CFD case as either converged, given the label "PASS", or not yet converged, given the label "EXTEND". The methodology and implementation of these models, along with the data used to train and validate the models, will be discussed in subsequent chapters.

2. Background

The two fields of machine learning and CFD both have their roots in the early 20th century. Two dimensional CFD calculations were first performed in the 1940's. In the 1960's, a more modern approach of discretizing a surface was implemented. However, it was not until the 1970's as computational power increased that there was a shift from solving the linearized potential equations to the non-linear, or full potential equations[4]. Similarly, machine learning theory was developed in the 1940's, evolving to a computer program for playing checkers in the 1950's. The field continued to advance into the 1970's, but was limited by inefficiencies of both the algorithms and computational power[5]. It wasn't until the 1990's that both fields exploded with the rapid advancement of computer technology.

Machine learning presents wonderful opportunities in CFD, and numerous applications of machine learning techniques have already been identified. Machine learning techniques have been utilized to guide the development of run matrices[6], optimize grid generation[7], predict physical quantities[8], and even develop new Reynolds Averaged Navier-Stokes (RANS) models[9].

2.0 CFD Convergence

Developing an adequate convergence criterion is a crucial step in any iterative method. CFD simulates complex physical processes, ranging from natural convection to multi-component, non-equilibrium hypersonic flows. A model is thought to be converged when the error in the physical quantity of interest has decreased below the required precision. An indicator of the current error is a residual value which measures the local imbalance of a conserved variable [https://www.engineering.com/story/3-criteria-for-assessing-cfd-convergence]. Since each cell of a grid will have its own residual, the root mean square (RMS) is often used to output a single value per iteration. Throughout the iterative process, the residuals are expected to progressively decay to smaller values up until they level out and substantial changes stop occurring[10]. As a rule of thumb, a drop of three orders of magnitude in the RMS residuals is the minimum level of convergence required for useful results[11].

Although tracking residuals can provide valuable insight, physical properties can also be important convergence indicators. One text book giving an example of biomass combustion modeling is Ansys Fluent recommends net flux imbalance should end in a result inferior to 1% of the smallest flux through the smallest inlet/outlet result[10]. Another conference journal investigating the efficiency of vaned diffuser of centrifugal compressor used a convergence criteria of mass flow rate discrepancy at the stage inlet equal to 0.001 kg/s, and temperature discrepancy at the exit equal to 0.1ºC[12]. For a more complete list of convergence criteria recommendations, see chapter 7 of [13].

2.1 Deep Learning

For CFD simulations that cannot automate termination of a case using the convergence criteria provided in section 2.1, it is often left to the practitioner's judgement. The simulation outputs may be too noisy or complex and thus require more complex algorithms. To classify a CFD case with more complex outputs, a deep learning model can be implemented. Deep learning is a type of machine learning that employs multiple layers of processing to extract higher level features in data. As a deep learning model is trained, it automatically detects features, also known as feature learning or representation learning, and can then classify each case.

2.1.1 Shallow Learning Representation

A shallow learning representation is utilized to provide a mathematical look under the hood of the deep learning model presented in this paper. A shallow learning model consist of an input layer, a single hidden layer, and an output layer, whereas deep learning models can be comprised of multiple inputs and hidden layers, forming the basis of a neural network. Figure 2.1 visualizes the shallow learning model discussed in the subsequent paragraphs.

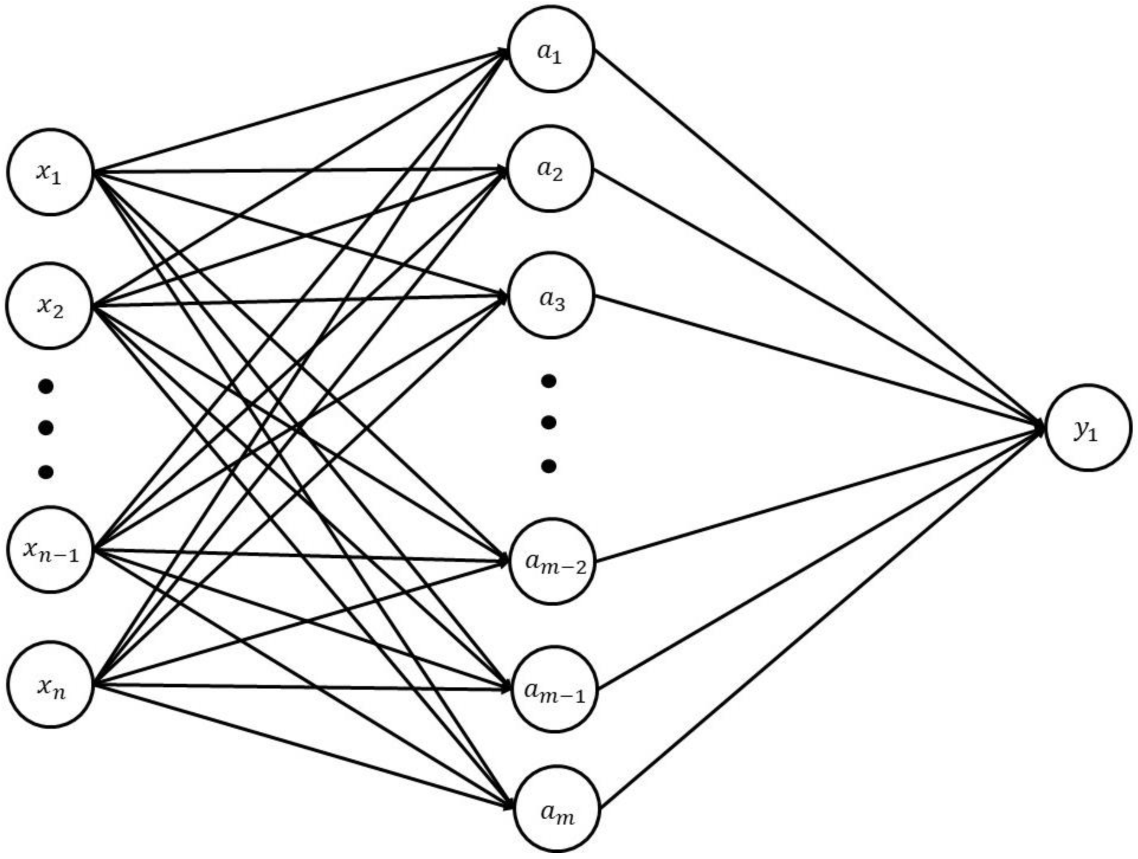


Figure 2.1 single hidden layer neural network.

The input layer is comprised of a vector of elements x_1 to x_n. For this paper's purposes, the input vector is the iterative history for a single aerodynamic coefficient from a single simulation.

The super script denotes which layer is being referenced, layer 0, 1, 2 corresponding to input, hidden, and output layer.

$$\vec{x}^{[0]} = (x_1, x_2, \dots, x_{n-1,} x_n) \tag{2.1}$$

Each neuron in the hidden layer takes the full input vector as an input, performs two operations, and outputs a single value. The first operation is each input element multiplied by a corresponding weight plus a corresponding bias. The second operation is an activation function, which determines the output of the neuron. For this example, the hidden layer utilizes a rectified linear unit (ReLu) activation function. These two steps for the hidden layer are presented in vectorized form in questions 2.2 and 2.3. A depiction of the ReLu activation function is presented in Figure 2.2.

$$\vec{z}^{[1]} = \overrightarrow{W}^{[1]}\vec{x}^{[0]} + \vec{b}^{[1]} \tag{2.2}$$

$$\vec{a}^{[1]} = \phi(\vec{z}^{[1]}) \tag{2.3}$$

$$Where, \qquad \phi(z) = \begin{cases} 0 \ \ if \ z \le 0 \\ z \ \ if \ z > 0 \end{cases} \tag{2.4}$$

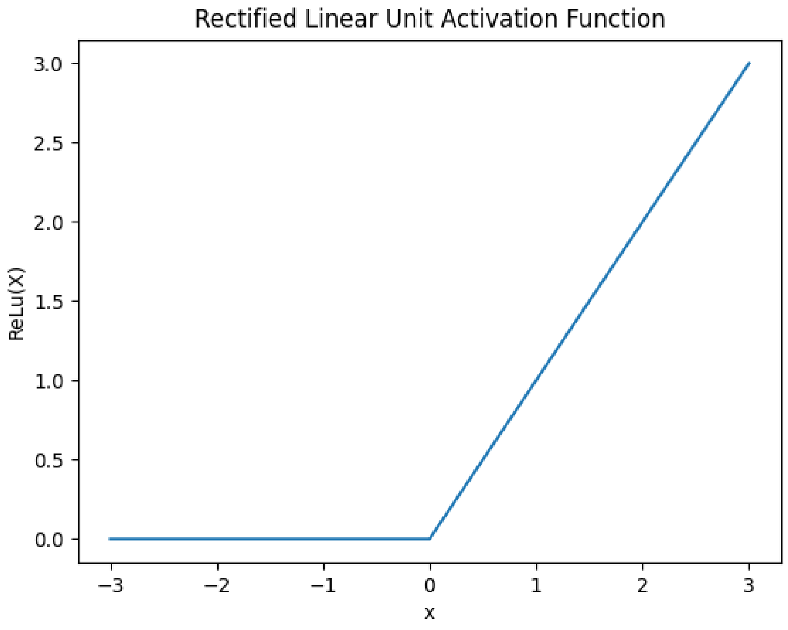


Figure 2.2. ReLu activation function graphed.

The number of neurons in the output layer will correspond to the number of classes. Therefore, a binary classification problem utilizes a single neuron. For this example, the single neuron implements a sigmoid activation function is used to return values between 0 and 1. Like the hidden layer, the inputs of the output layer are first multiplied by the weights in the output layer. A depiction of the sigmoid activation function is presented in Figure 2.3.

$$z^{[2]} = \overrightarrow{W}^{[2]}\vec{a}^{[1]} + \vec{b}^{[2]} \tag{2.5}$$

$$\vec{a}^{[2]} = \sigma(\vec{z}^{[2]}) \tag{2.6}$$

$$Where, \qquad \sigma(z) = \frac{1}{1 + e^{-x}} \tag{2.7}$$

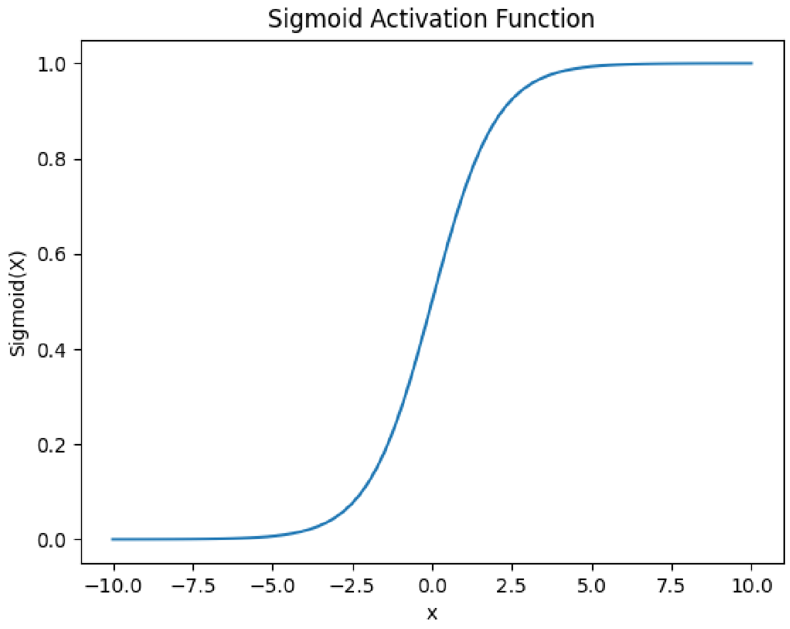


Figure 2.3. sigmoid activation function graphed.

For a binary classification problem, one label would correlate to outputs greater than 0.5, while the other label would correlate to values less than or equal to 0.5.

$$\begin{cases} Label_a \ if \ \sigma(\vec{z}^{[2]}) \leq 0.5 \\ Label_b \ if \ \sigma(\vec{z}^{[2]}) > 0.5 \end{cases} \tag{2.8}$$

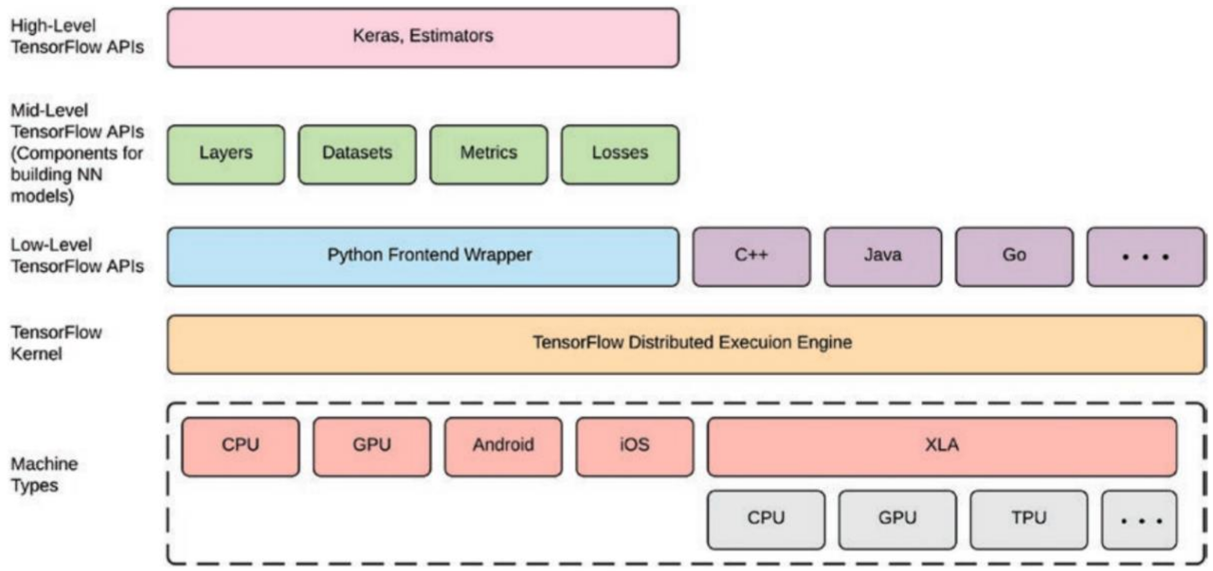
A loss function is used to ascertain how close the output layer's output was to the correct answer. It does so by comparing the output to the correct label. Equation 2.9 presents the binary cross-entropy loss function. In this equation, y represents the true value, 0 or 1, and p represents the neural network output.

$$H = -(y \log(p) + (1 - y) \log(1 - p)) \tag{2.9}$$

Training a neural network often refers to adjusting the weights and biases to minimize the loss function. This can be done through optimization methods such as gradient descent.

2.1.2 TensorFlow API

TensorFlow[14] is an end-to-end, open-source artificial intelligence (AI) framework which provides high level application programming interfaces (API). Machine learning models built with TensorFlow use a hierarchy of APIs. Low-level APIs perform the bulk of mathematical operations and are implemented behind the scenes in high-level APIs. Mid-level APIs are the building blocks of a model, such as data transformations, neuron layers, and loss functions. High level APIs contain many of the tools needed for building models[15].


High-Level TensorFlow APIs
Keras, Estimators
Mid-Level TensorFlow APIs (Components for building NN models)
Layers
Datasets
Metrics
Losses
Low-Level TensorFlow APIs
Python Frontend Wrapper
C++
Java
Go
• • •
TensorFlow Kernel
TensorFlow Distributed Execuion Engine
Machine Types
CPU
GPU
Android
iOS
XLA
CPU
GPU
TPU
• • •
Figure 2.4. TensorFlow API hierarchy [15]

The high-level API, Keras, specializes in deep learning, and is therefore employed in each model. Within Keras, there are two main APIs for building models. First is a sequential API, which as the name may imply, each layer is built in a sequence one atop another. This is a more intuitive API, and only takes a single input and gives a single output. The second method is a functional API. The functional API is more flexible and allows branching and merging of layers, allowing for multi-input models.

2.1.3 Applied TensorFlow

TensorFlow is a powerful and popular machine learning framework that has been widely adopted in industry. Although machine learning algorithms have not previously been applied to classifying CFD convergence, there are similar model types and structures that can suggest best practices.

The Modified National Institute of Standards and Technology (MNIST) dataset consists of 70,000 28x28 pixel images of handwritten digits 1 through 10 and is commonly used as an introduction to machine learning. A multi-class convolutional neural network (CNN) is used to extract feature data from each image and properly classify the digit. A study seeking to demonstrate TensorFlow’s capabilities examined six different activation functions for the MNIST dataset. An activation function defines the output of a neuron and can be thought of as the on and off switch for a single neuron. For 60,000 training samples and 10,000 testing samples, the ReLu activation function provided the best accuracy, 98.43%[16].

Similarly, another study sought to use a CNN to classify 3670 224x224 RGB images into five classes: rose, daisy, dandelion, sunflower, and tulip. The models utilized the Keras application MobileNet, a general architecture designed to maximize CNN accuracy while limiting computational resources. This study achieved greater than 90% classification accuracy for each flower, and demonstrated that model size influences accuracy[17].

Lastly, a study used a TensorFlow deep learning neural network to predict asthma severity in patients. The HCUP National Inpatient Sample 2011 Database and the Medical Information Mart for Intensive Care III database were used for training the model. The key features included the patient's age, number of chronic conditions, admission month, length of stay, and number of comorbidities. Using a three-layer deep learning neural network, the national database achieved an accuracy of 86%, and the local hospital database achieved in the lower 90's. The same model was used for both datasets, and differences in accuracy are attributed to local environmental influences being better features in the hospital dataset[18].

3. Binary Classification Model Development

It is important to first define the terms associated with the model and its architecture. Each model is a supervised binary classification deep learning model with either single or multiple inputs depending on the Keras API. Breaking each part of the description down, supervised learning is when feature data comes with an associated label, meaning the model is training while knowing the correct answer. Binary classification requires that each sample can only be labeled as one of two groups. Deep learning refers to a multi-layer, iterative machine learning model that uses back propagation to adjust internal parameters to minimize a loss function. In TensorFlow, the number of iterations a model implements are known as epochs. Finally, clarification is needed when discussing model inputs. The sequential API takes a single input tensor, where each dimension above two is a feature variable. The functional API has multiple input branches, where each input is a single feature variable tensor.

Three models were created to initially develop best practices and model architecture: a single feature variable sequential API model; a multi-feature variable sequential API model; and a multi-feature variable function API model. The KerasTuner[19] framework is implemented on the third model to demonstrate hyperparameter optimization.

3.0 Split Input Data

The dataset used to initially determine best practices is different form the input data used to train the implemented model. Because the quality of the input data is crucial, the proper labeling was determined by surveying a professional practitioner. The survey is quite time intensive and therefore initial model development was performed in parallel with a pseudo dataset. The results of the survey and the final model are discussed in chapter 5.

The pseudo dataset used to train and validate each deep learning model consists of 600 samples, 300 converged cases and 300 not yet converged cases. Each coefficient from a sample represents a feature variable and the input features are the iterative history for each coefficient. The coefficients used as feature variables are the axial force coefficient (CA), side force coefficient (CY), normal force coefficient (CN), and pitching moment coefficient (CLM). 300 cases varying Mach number and angle of attack were run using the CFD software FUN3D. Each case used a simplified geometry, slightly reminiscent of the Atlas V-401, referred to as the common launch vehicle (CLV). The geometry is depicted in figure 3.1.

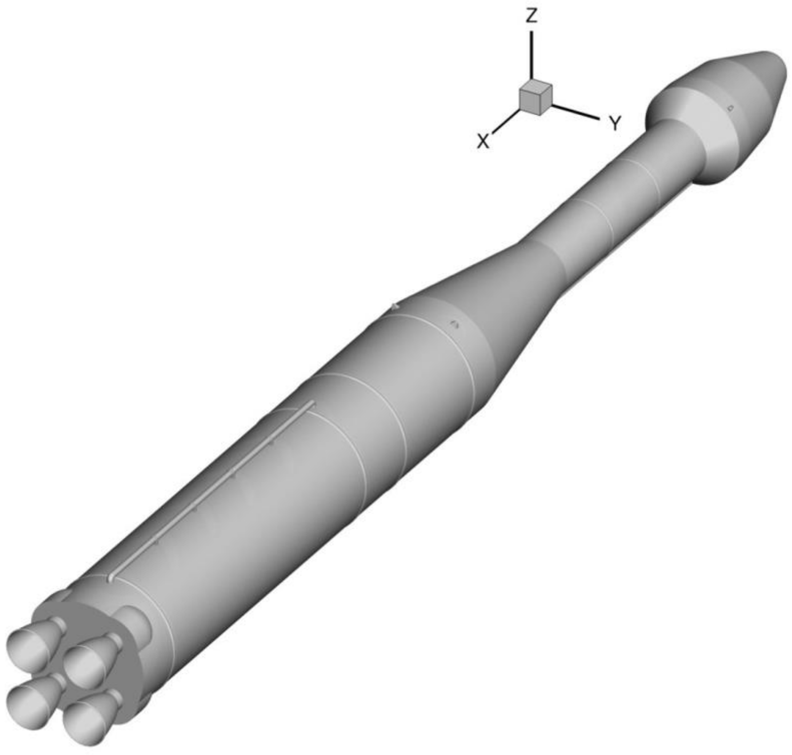


Figure 3.1 common Launch Vehicle

Each of the 300 cases are assumed to be correctly labeled as PASS. The last 1000 iterations of each case are used as features and labeled "PASS". To create the not yet converged cases, iterations 800 to 1800 were extracted and given the label "EXTEND". This assumption is sufficient for comparing model architectures, but, after further inspection of the input data, and confirmed later in the survey results, this assumption would have been inappropriate for the training the final model. The features for five cases are depicted for each feature variable in Figures 3.2 through 3.5.

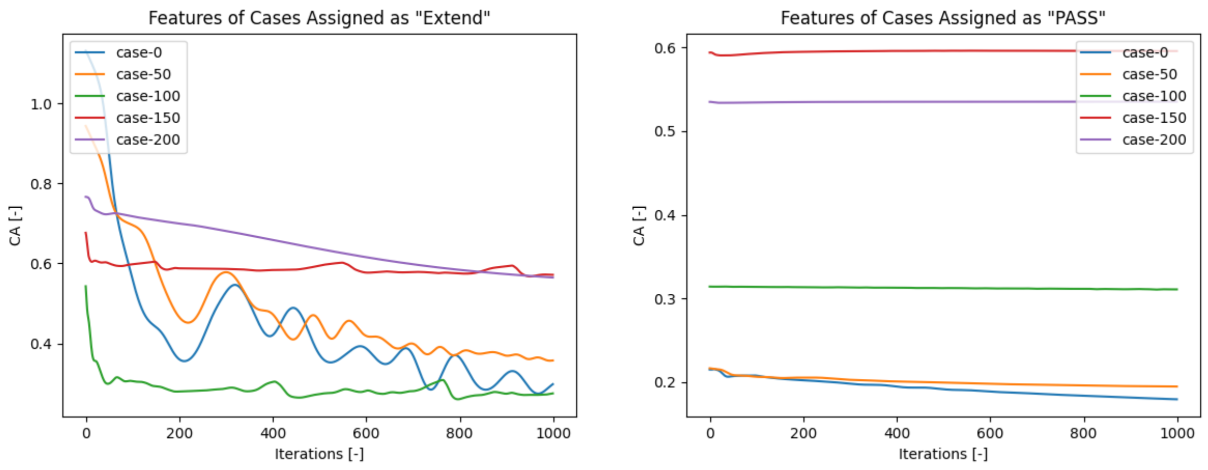


Figure 3.2. comparison between label data for CA

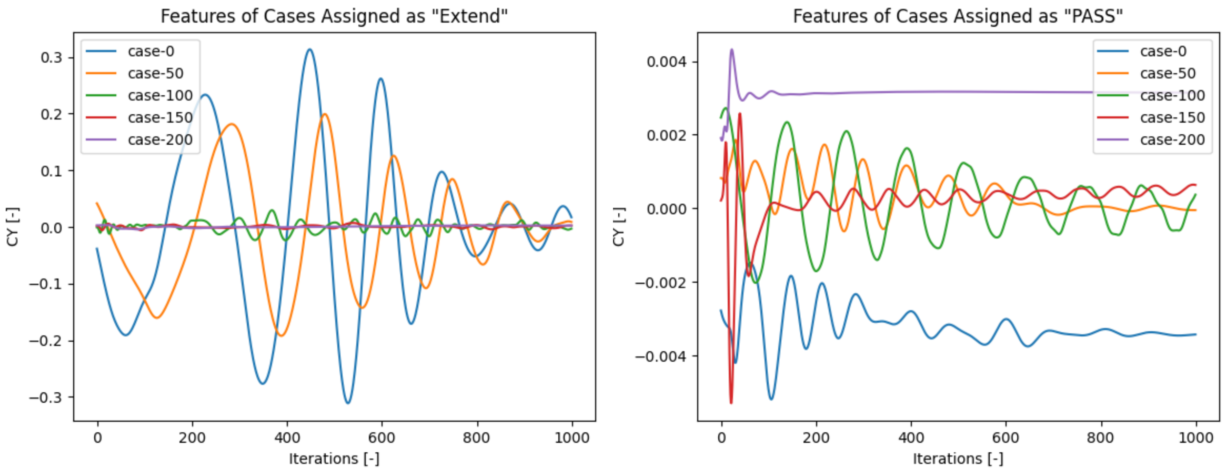


Figure 3.3 comparison between label data for CY

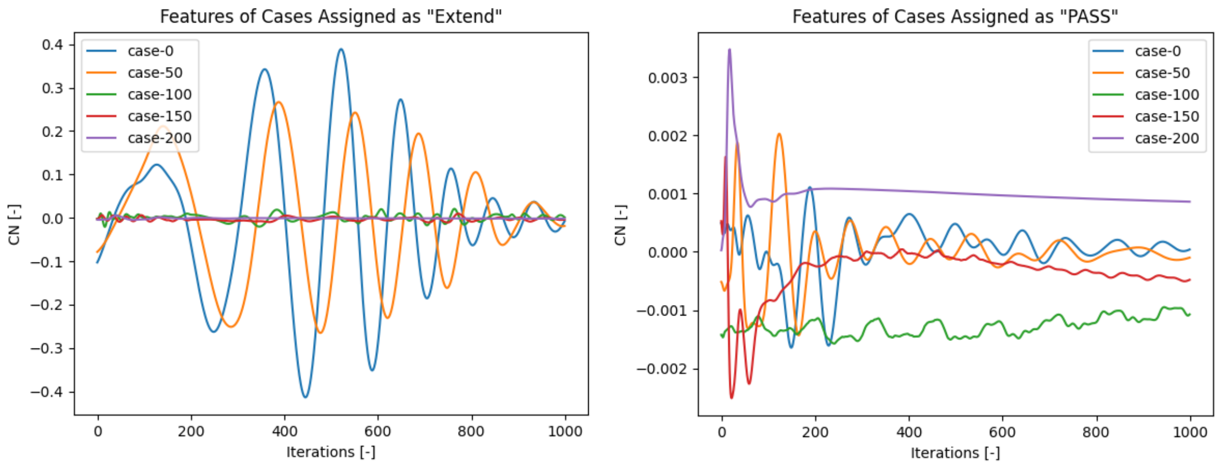


Figure 3.4 comparison between label data for CN

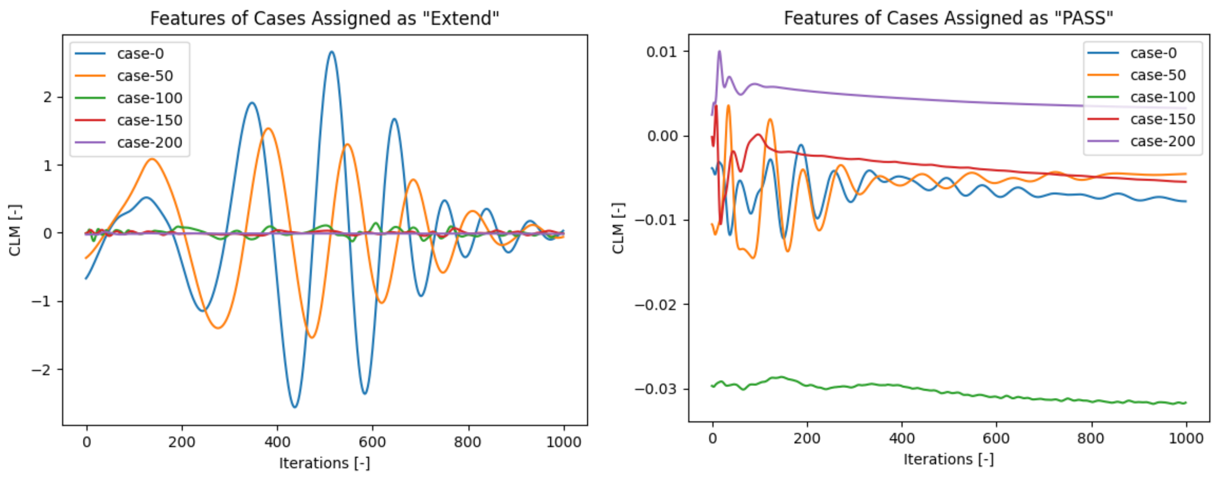


Figure 3.5 comparison between label data for CLM

Each force coefficient depicts less variation in the feature data associated with the PASS label, particularly in the latter half of the case features. For cases 0 and 50, their local mean continues to drift in the latter iterations, and an argument can be made that these cases have not yet converged. Alternatively, cases 100 and 150’s features for the EXTEND label appear more similar to converged cases.

3.1 Single Feature Variable Sequential Model

The simplest of the three models is the single feature variable binary classification model using the Keras sequential API. This model takes a single tensor input containing only one force coefficient. Given the relatively low number of training parameters (summation of layer input features multiplied by neuron density), this model completes training for epochs on the order of 10^1. To train and validate the model, 80% of the dataset was devoted to training, with the other 20% being used for validation. The model summary and validation results are provided in figure 3.6.

The first dense layer is the input layer, and its output shape is the number of neurons contained within that layer. The next two layers are referred to as hidden layers and are where neurons receive weighted inputs and output values according to activation functions. A rectified linear activation function (ReLu) is implemented in each hidden layer. The ReLu activation function returns the input as is if the input is positive and returns zero if the input is negative and is the most popular activation function for deep neural networks. The final layer is the output layer and uses the sigmoid activation function to produce a single output value between 0 or 1. A threshold of 0.5 determines if the sample is classified as either PASS or EXTEND.

Table 3-1. Multi-Feature Variable Binary Classification Model Summary

```
            Model: "single feature variable sequential API"
_____________________________________________________________
  Layer (type)             Output Shape              Param #
=============================================================
 dense_0 (Dense)            (None, 50)                75050

 dense_1 (Dense)            (None, 25)                1275

 dense_2 (Dense)            (None, 10)                260

 dense_3 (Dense)            (None, 1)                 11

=============================================================
                   Total params: 76,596
                 Trainable params: 76,596
                  Non-trainable params: 0
_____________________________________________________________

   CA: validation loss, validation accuracy: [0.3647, 0.9167]
                Execution time in seconds: 9.77
   CY: validation loss, validation accuracy: [0.3928, 0.9000]
                Execution time in seconds: 9.37
    CN validation loss, validation accuracy: [0.4053, 0.8667]
                Execution time in seconds: 11.10
  CLM validation loss, validation accuracy: [0.5372, 0.7500]
                Execution time in seconds: 10.10
```

How far the output value is from either 0 or 1 influences the output of the model's binary cross entropy loss function. The objective function for this model is binary accuracy. The general structure of ReLu hidden layers followed by a sigmoid output layer, with a binary cross entropy loss function and binary accuracy cost function is consistent across all three models.

3.2 Multi-Feature Variable Sequential Model

The multi-feature binary classification model is constructed very similar to the single-feature model. Instead of an input tensor containing only a single feature variable, now its dimensionality has increased to pass through four feature variables: CA, CY, CN, CLM. The trainable parameters correlate linearly with the number of feature variables. To cope with this added complexity, epochs are needed on the order of 10^3.

Table 3-2. Multi-Feature Variable Binary Classification Model

Model: "multi-feature variable sequential API"

Layer (type)	Output Shape	Param #
dense_0 (Dense)	(None, 64)	256,000
dense_1 (Dense)	(None, 32)	2080
dense_2 (Dense)	(None, 16)	528
dense_3 (Dense)	(None, 8)	136
dense_4 (Dense)	(None, 1)	9

Total params: 258,753
Trainable params: 258,753
Non-trainable params: 0

test loss, test acc: [0.0258, 1.0]
Execution time in seconds: 335.94

The test accuracy of this model exemplifies the need for multiple feature variables. The highest accuracy achieved by the singe feature variable sequential model was 91.67%; whereas 100% accuracy was achieved by the multi-feature variable sequential model. The tradeoff comes in the greater execution time of the multi-feature variable sequential model.

3.3 Multi-Input Functional Model

Keras functional API allows for network branches, which take single feature variable tensors as inputs, processes the features through multiple hidden layers, and then merges the outputs back into a single tensor prior to the output layer.

The multi-feature variable functional model outperformed the multi-feature variable sequential model in every metric. Available in figure 3.9, the multi-feature variable functional model similarly achieved perfect accuracy, but in half the time for the same epochs and achieved a test loss two orders of magnitude less than the multi-feature variable sequential model. For these reasons, the function API is implemented for further model optimization

Table 3-3. Multi-Input Binary Classification Model Summary

```
                         Model: “multi-input functional API "
________________________________________________________________________
         Layer (type)              Output Shape      Param #   Connected to
========================================================================
      CA (Input Layer)              [(None, 1000)]       0          []
      CY (Input Layer)              [(None, 1000)]       0          []
      CN (Input Layer)              [(None, 1000)]       0          []
      CLM (Input Layer)             [(None, 1000)]       0          []
      dense (Dense)                (None, 128)       128128     ['CA[0][0]']
      dense_4 (Dense)              (None, 128)       128128     ['CY[0][0]']
      dense_8 (Dense)              (None, 128)       128128     ['CN[0][0]']
      dense_12 (Dense)             (None, 128)       128128     ['CLM[0][0]']
      dense_1 (Dense)              (None, 40)         5160      ['dense[0][0]']
      dense_5 (Dense)              (None, 40)         5160      ['dense_4[0][0]']
      dense_9 (Dense)              (None, 40)         5160      ['dense_8[0][0]']
      dense_13 (Dense)             (None, 40)         5160      ['dense_12[0][0]']
      dense_2 (Dense)              (None, 64)         2624      ['dense_1[0][0]']
      dense_6 (Dense)              (None, 64)         2624      ['dense_5[0][0]']
      dense_10 (Dense)             (None, 64)         2624      ['dense_9[0][0]']
      dense_14 (Dense)             (None, 64)         2624      ['dense_13[0][0]']
      dense_3 (Dense)              (None, 8)           520      ['dense_2[0][0]']
      dense_7 (Dense)              (None, 8)           520      ['dense_6[0][0]']
      dense_11 (Dense)             (None, 8)           520      ['dense_10[0][0]']
      dense_15 (Dense)             (None, 8)           520      ['dense_14[0][0]']
      concatenate (Concatenate)  (None, 32)             0      ['dense_3[0][0]',
                                                                'dense_7[0][0]',
                                                                'dense_11[0][0]',
                                                                'dense_15[0][0]']
      dense_16 (Dense)             (None, 36)         1188      ['concatenate[0][0]']
      dense_17 (Dense)             (None, 15)          555      ['dense_16[0][0]']
      dense_18 (Dense)             (None, 9)           144      ['dense_17[0][0]']
      dense_19 (Dense)             (None, 12)          120      ['dense_18[0][0]']
      dense_20 (Dense)             (None, 1)            13      ['dense_19[0][0]']

========================================================================
                          Total params: 547,748
                        Trainable params: 547,748
                         Non-trainable params: 0
                  ____________________________________________

                    test loss, test acc: [0.0009, 1.0]
                    Execution time in seconds: 170.20
```

3.4 Hyperparameter Optimization

Each of the three models previously presented were constructed through a heavily manual iterative process of educated tinkering. To better optimize a TensorFlow model, the KerasTuner optimization framework should be utilized. The KerasTuner framework introduces hyperparameters, allowing for internal model parameters, such as number of hidden layers or activation functions, to become variables. Eleven hyperparameters were given a small range of two to four values. Given many hyperparameters, an optimization study with adequate spatial resolution is extremely computationally expensive.

To mitigate computational cost, a Bayesian optimization approach is used to optimize the set of hyperparameters. Bayesian optimization is a probabilistic approach that considers already tested combinations to sample the next combination. Randomly generated samples can be used as initial training data for Bayesian optimization. If no amount of initial random seeds is specified, TensorFlow will randomly generate samples 3 times the dimensionality of the hyperparameter space. An example of a set of hyperparameters in presented in table 3.1. The best performing model is tracked as trials progress.

Table 3-4. Example of hyperparameter combination when beginning Bayesian optimization.

Search: Running Trial #1		
Value	**Best Value so Far**	**HyperParameter**
192	?	input density
3	?	n_layers_branch
24	?	input_merge_density
3	?	n_layers_merge
40	?	dense_0_units_branch
8	?	dense_1_units_branch
24	?	dense_2_units_branch
12	?	dense_0_units_merge
12	?	dense_1_units_merge
18	?	dense_2_units_merge
0.001	?	learning_rate

Each trial run is stored in event files that can be viewed with TensorBoard. TensorBoard is a visualization tool for tracking metrics and evaluating sampling coverage, and was used to confirm sufficient epochs. The Python code for this model can be found in Appendix D.

3.5 Model Sensitivity to Number of Features

Determining the number of features and feature variables is a difficult question. To offer a brief insight in this problem, a study was performed to evaluate the sensitivity of the model's performance to the number input features per feature variable. Three feature amounts are compared in Figures 3.10 and 3.11. The final 500, 1000, 1500 consecutive iterations for each force coefficient comprise PASS cases, while iterations 800:[1300,1800,2300] comprise EXTEND cases. For each amount of input features, 75 hyperparameter combinations were evaluated using Bayesian optimization. To ensure each model converged, an additional 200 training epochs were used for each additional 500 features. 300 training epochs were used for the initial 500 features. Increasing the number of epochs results in the computational time required for training a single hyperparameter combination more than doubles between the 500 and 1500 features.

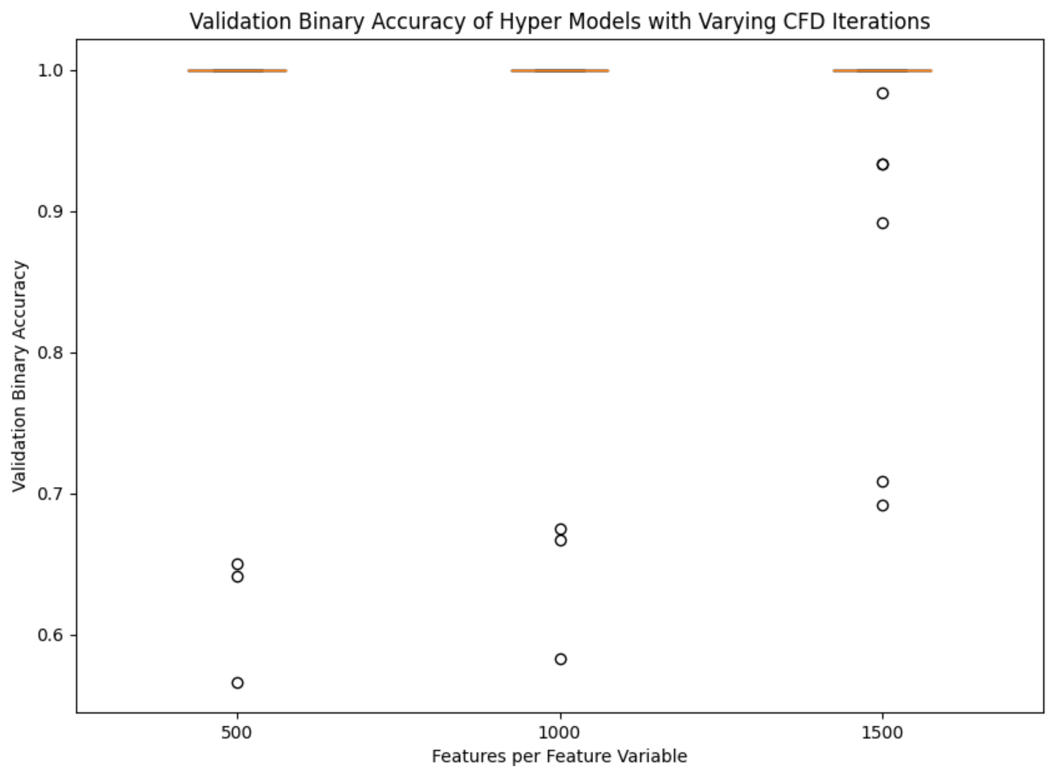


Figure 3.6 Validation Binary Accuracy of Models during Hyperparameter Optimization

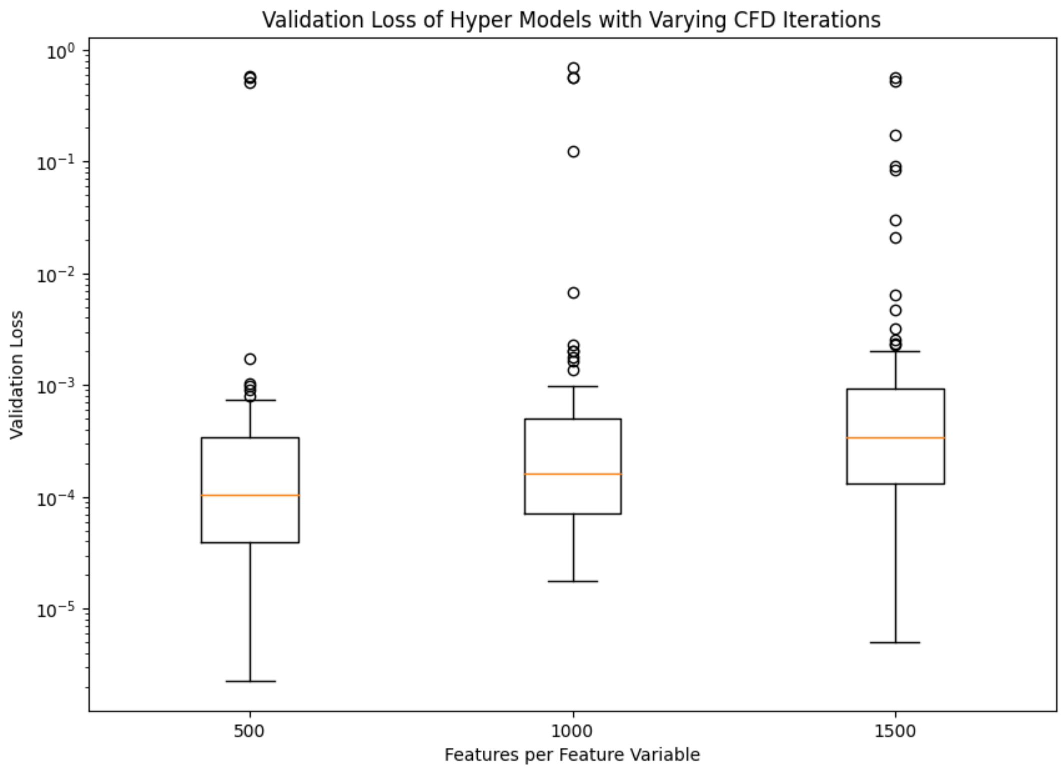


Figure 3.7 Validation Binary Accuracy of Models during Hyperparameter Optimization

As seen in Figures 3.6 and 3.7, each feature number variation achieved excellent validation accuracy and on average excellent validation loss. As the number of input features increase, the validation loss increases for more model architectures. This study insinuates that for this training and validation dataset an optimized model is not particularly sensitive to the number of input features. However, this result can be attributed to the split input data referenced in ection 3.0. Splitting each CFD case introduces unique differences that may not exist in a higher quality training dataset. Knowledge pertaining to the nature of the CFD cases and the physical quantities of interest should guide the type of input feature variables and number of features per feature variable. Half of the 300 cases are below Mach 1.10 and contain unsteady behavior in their iterative history. Considering this, 500 iterations may be insufficient for capturing the transient frequency.

4. Simple CFD Convergence Criteria

A common convergence criterion is to monitor physical quantities of interest for asymptotic behavior. If the quantity of interest does not change by a specified amount over a specified number of iterations, then the simulation has converged on a solution. The criteria can be expanded to track multiple quantities of interest with varying required precision.

Two simple convergence criteria are developed in this chapter and are later used as a comparison against an optimized multi-feature variable functional model in chapter 6. The first is a standard algorithm determining the maximum change across a specified number of iterations. The second algorithm also checks for asymptotic behavior, but first smooths the iterative history to account for oscillations.

4.0 Asymptotic Criterion

Although one of the simplest CFD convergence criteria, checking for asymptotic behavior is one of the most widely implemented algorithms. Both industry giants, Siemen's STAR-CCM+ and Ansys Fluent have integrated the monitor into their respective programs. To check for asymptotic behavior, simply determine the maximum and minimum values in the desired iteration range. If the difference between these values exceeds the required precision, then the simulation continues.

A change of less than 0.001 over the last 500 iterations for every force coefficient is deemed sufficient for classifying a solution as converged.

$$\begin{cases} PASS & if \quad |\max(iterative\ range) - \min(iterative\ range)| < 0.001 \\ EXTEND & if \quad |\max(iterative\ range) - \min(iterative\ range)| \geq 0.001 \end{cases} \tag{4.1}$$

The precision and iteration window were chosen to roughly match the PASS/EXTEND ratio of the survey results.

4.1 Asymptotic Criterion with Feature Smoothing

For subsonic and transonic Mach numbers, force coefficient plots produced in FUN3D can contain oscillations as a result of actual physical unsteadiness. These oscillations can have a relatively constant mean and be classified as converged solutions, but the large amplitude of the oscillations prevent a standard asymptotic algorithm from being applied. To remedy this, the iterative history is smoothed to allow for an asymptotic algorithm to be applied. An example of feature smoothing is depicted in Figure 4.1.

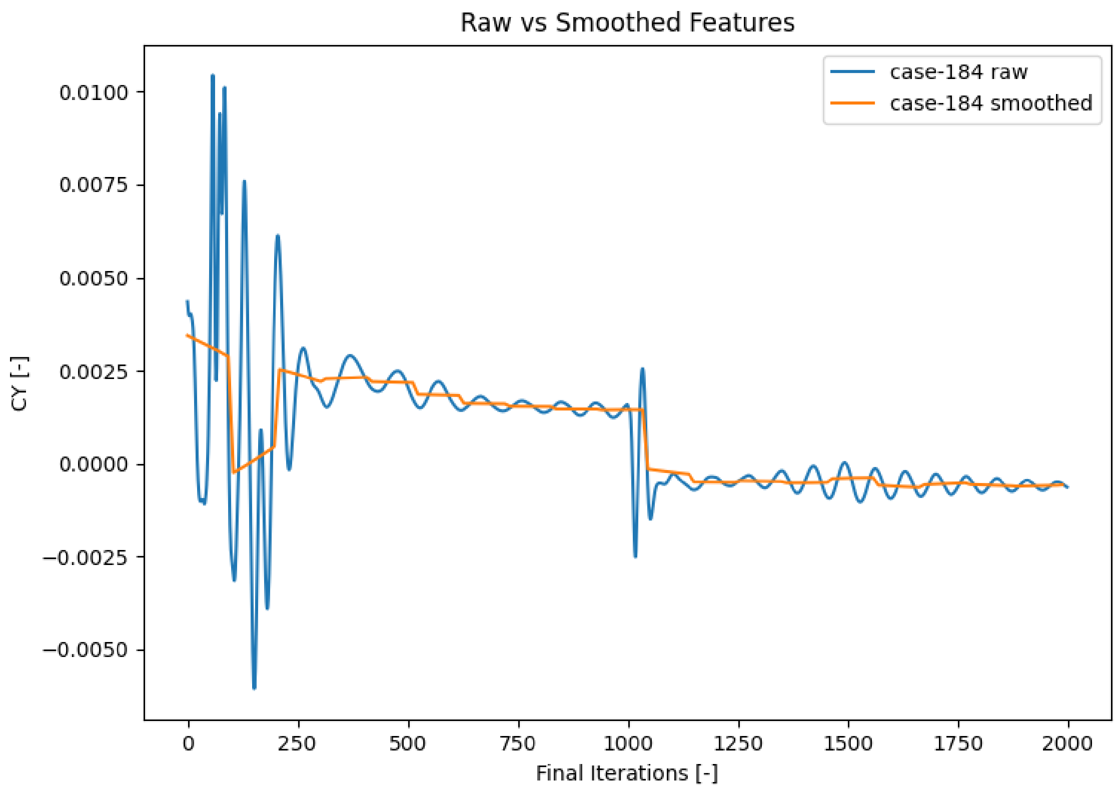


Figure 4.1 Feature Smoothing of an Oscillatory Iterative History

The feature smoothing was performed by stepping though averaging windows of 100 iterations. The first value would be the average of iterations 0 to 100, the second value would be the average of iterations 1 to 101, and so on and so forth. Potential limitations of this method are if the frequency is greater than the specified window, then the smoothed features could be oscillatory. Once the features have been smoothed, simply apply the previous asymptotic check.

5. Final Machine Learning Model

Referencing the best practices learned and studies performed in chapter 3, a multi-input functional model which samples 1000 iterations was selected. The following sections describe the training and validation dataset, the final model architecture, and preliminary convergence criteria comparisons using the training and validations dataset.

5.0 Training and Validation Dataset

To create the training and validation dataset, an expert practitioner of FUN3D was surveyed via a dash app hosted by Heroku. The survey consists of the 300 CFD cases referenced in section 3.0. Each coefficient's iterative history can be viewed interactively using the Python graphing library Plotly. This interface allowed the expert full control to ensure a presented range or scale would not bias their decision. No information about the case was provided. The full survey can be found at https://jacket-sculpture-silica-g8mp.herokuapp.com/. Based on the expert's classification, 117 out of the 300 cases were labeled as PASS. From the first half of the cases, where the Mach number is 1.10 or less, 38 cases were labeled as PASS. A random 80% of the dataset was used for training, and 20% was used for validation and optimizing the hyperparameters.

5.1 Final Model Architecture

The final model architecture was selected from the model sensitivity study conducted in section 3.5 and is depicted in figure 5.1. The model selected achieved the lowest validation loss while tuning the hyperparameters. Although this was expected to be the final model, for reasons discussed in chapter 6, a second model that considered a range of 2000 iterations was trained and optimized. The most notable change between these two models was an increase in the input dense layer from 192 neurons to 500 neurons. The second model was added later in the project, and results in chapter 5 do not depict the second model. Therefore, all subsequent references to machine learning model in this chapter refer to the machine learning model which samples 1000 iterations.

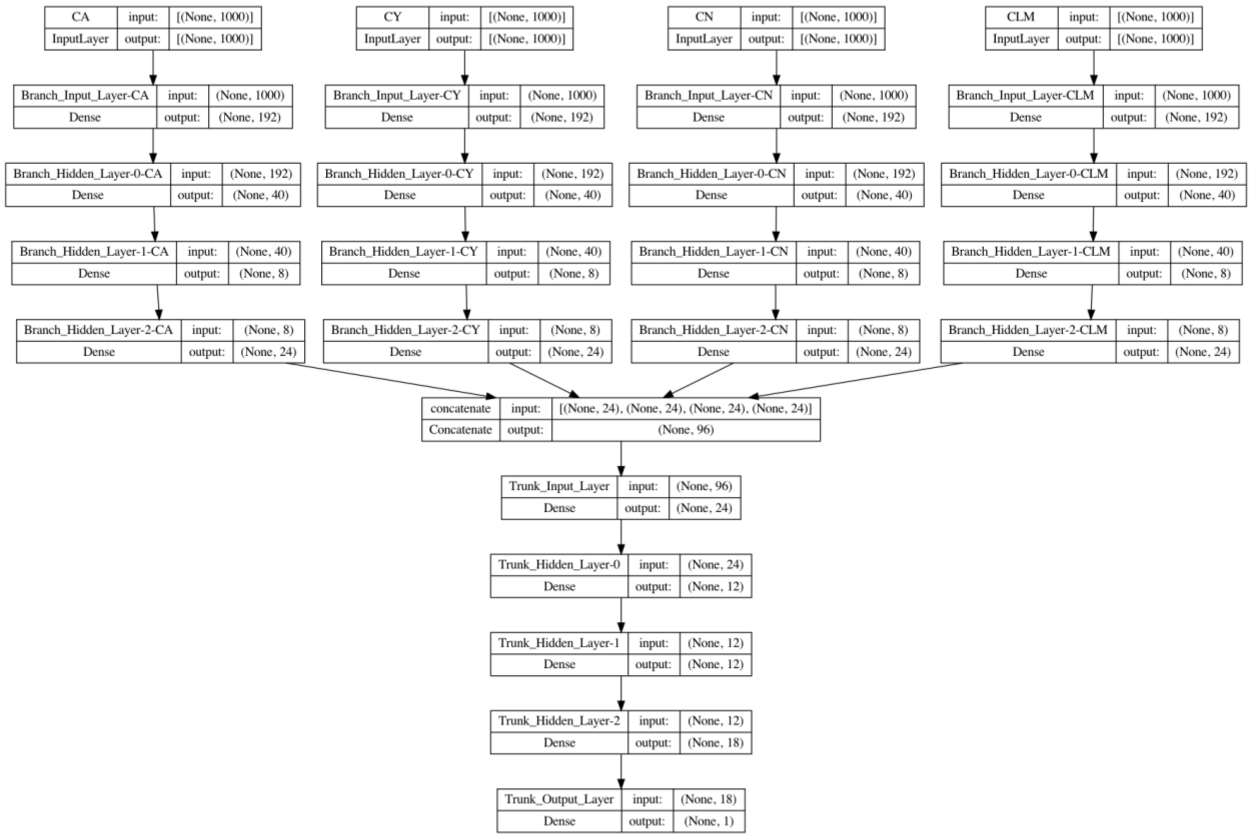

CA | input: | [(None, 1000)]
InputLayer | output: | [(None, 1000)]
CY | input: | [(None, 1000)]
InputLayer | output: | [(None, 1000)]
CN | input: | [(None, 1000)]
InputLayer | output: | [(None, 1000)]
CLM | input: | [(None, 1000)]
InputLayer | output: | [(None, 1000)]
Branch_Input_Layer-CA | input: | (None, 1000)
Dense | output: | (None, 192)
Branch_Input_Layer-CY | input: | (None, 1000)
Dense | output: | (None, 192)
Branch_Input_Layer-CN | input: | (None, 1000)
Dense | output: | (None, 192)
Branch_Input_Layer-CLM | input: | (None, 1000)
Dense | output: | (None, 192)
Branch_Hidden_Layer-0-CA | input: | (None, 192)
Dense | output: | (None, 40)
Branch_Hidden_Layer-0-CY | input: | (None, 192)
Dense | output: | (None, 40)
Branch_Hidden_Layer-0-CN | input: | (None, 192)
Dense | output: | (None, 40)
Branch_Hidden_Layer-0-CLM | input: | (None, 192)
Dense | output: | (None, 40)
Branch_Hidden_Layer-1-CA | input: | (None, 40)
Dense | output: | (None, 8)
Branch_Hidden_Layer-1-CY | input: | (None, 40)
Dense | output: | (None, 8)
Branch_Hidden_Layer-1-CN | input: | (None, 40)
Dense | output: | (None, 8)
Branch_Hidden_Layer-1-CLM | input: | (None, 40)
Dense | output: | (None, 8)
Branch_Hidden_Layer-2-CA | input: | (None, 8)
Dense | output: | (None, 24)
Branch_Hidden_Layer-2-CY | input: | (None, 8)
Dense | output: | (None, 24)
Branch_Hidden_Layer-2-CN | input: | (None, 8)
Dense | output: | (None, 24)
Branch_Hidden_Layer-2-CLM | input: | (None, 8)
Dense | output: | (None, 24)
concatenate | input: | [(None, 24), (None, 24), (None, 24), (None, 24)]
Concatenate | output: | (None, 96)
Trunk_Input_Layer | input: | (None, 96)
Dense | output: | (None, 24)
Trunk_Hidden_Layer-0 | input: | (None, 24)
Dense | output: | (None, 12)
Trunk_Hidden_Layer-1 | input: | (None, 12)
Dense | output: | (None, 12)
Trunk_Hidden_Layer-2 | input: | (None, 12)
Dense | output: | (None, 18)
Trunk_Output_Layer | input: | (None, 18)
Dense | output: | (None, 1)

Figure 5.1. Multi-Input Binary Classification Model Structure

5.2 Test and Validation Dataset Results

For results below, the machine learning model was asked to make a prediction for 100% of the 300 cases used for training and validation. This was done to provide a full comparison between the machine learning model and the simple algorithms.

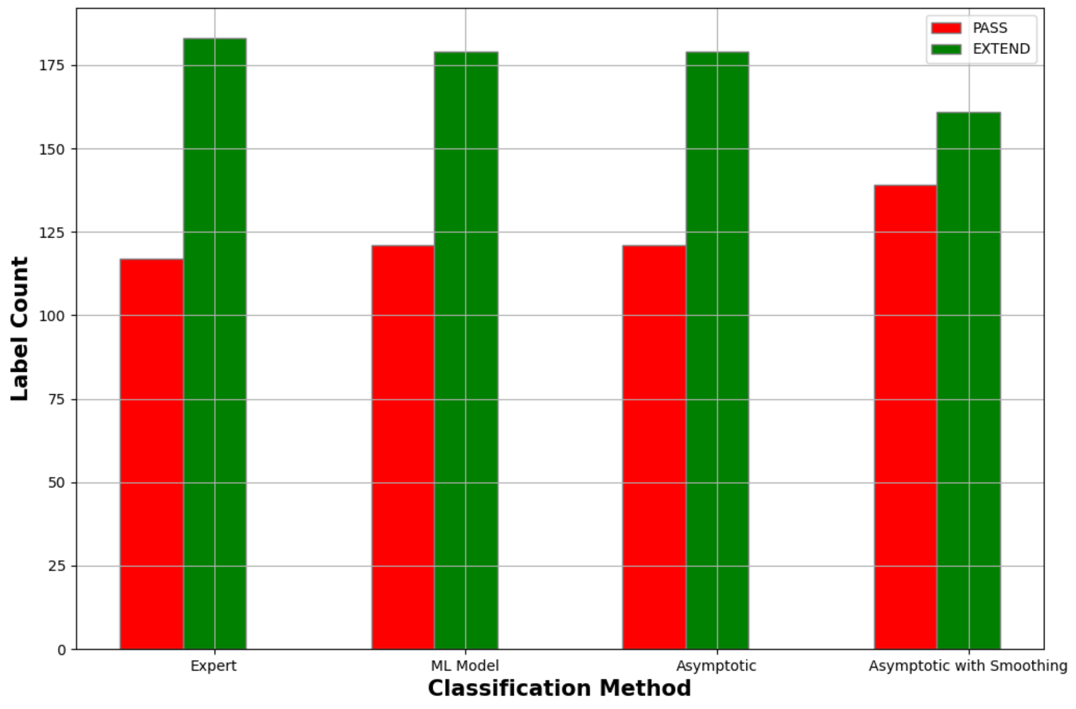


Figure 5.2. Classification Results

Although the bar plot may infer that the models have similar predictions, table 5-1 tells otherwise. The expert disagreed with the machine learning model on 4% of cases and the asymptotic model on 33% of cases. Even the asymptotic model and the asymptotic model with feature smoothing disagreed on 17% of cases. The bottom row of the table states how many cases were labeled PASS by only that model.

Table 5-1. Disagreements between Labels

	Expert	ML Model	Asymptotic	Asymptotic w/Smoothing
Expert	0	12	98	86
ML Model	12	0	92	88
Asymptotic	98	92	0	52
Asymptotic w/Smoothing	86	88	52	0
Accuracy %	N/A	96.0%	67.3%	71.3%
Unique PASS	0	2	6	10

Below is a case example for each of the predictive models' unique PASS label.

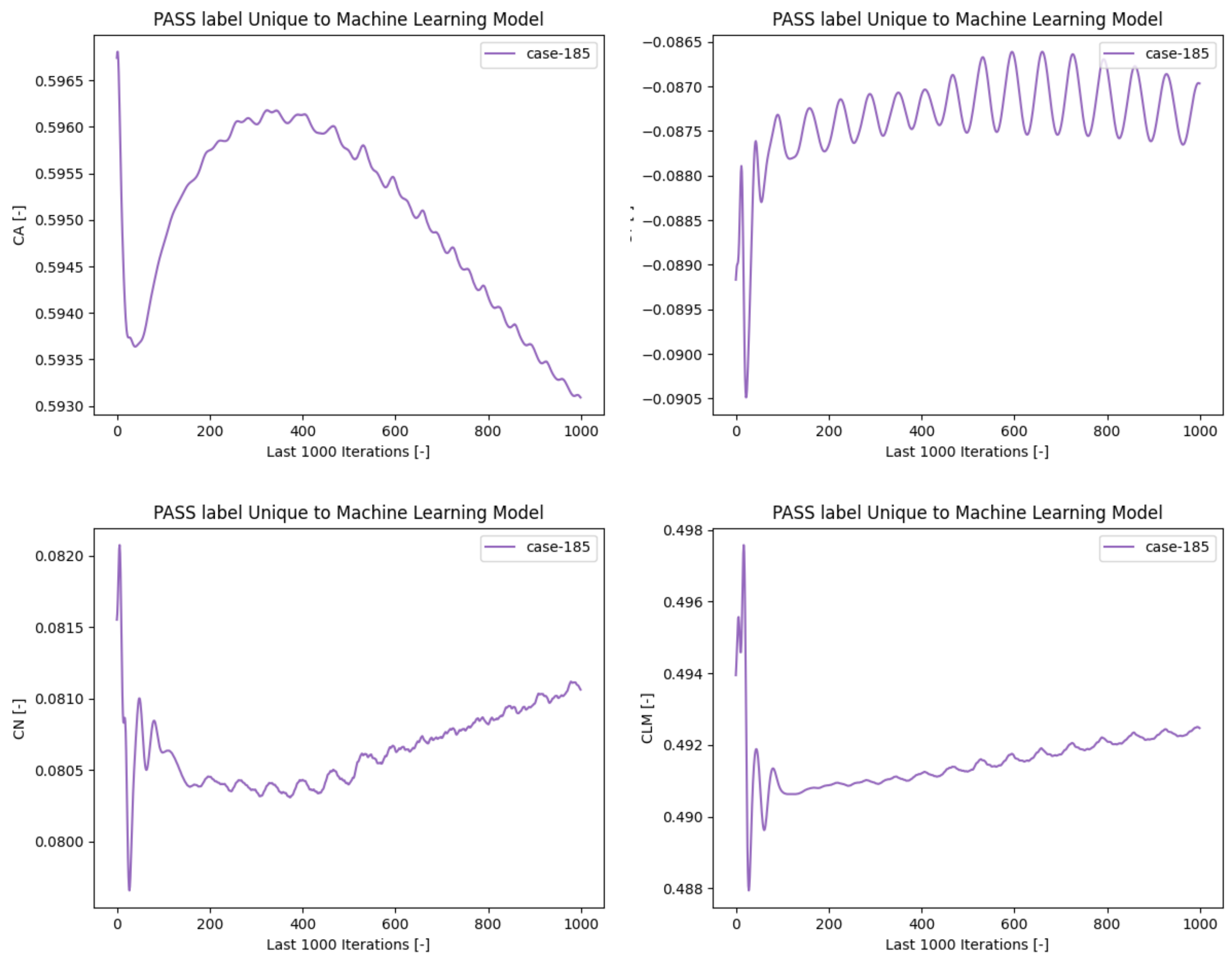


Figure 5.3. PASS labels unique to machine learning model.

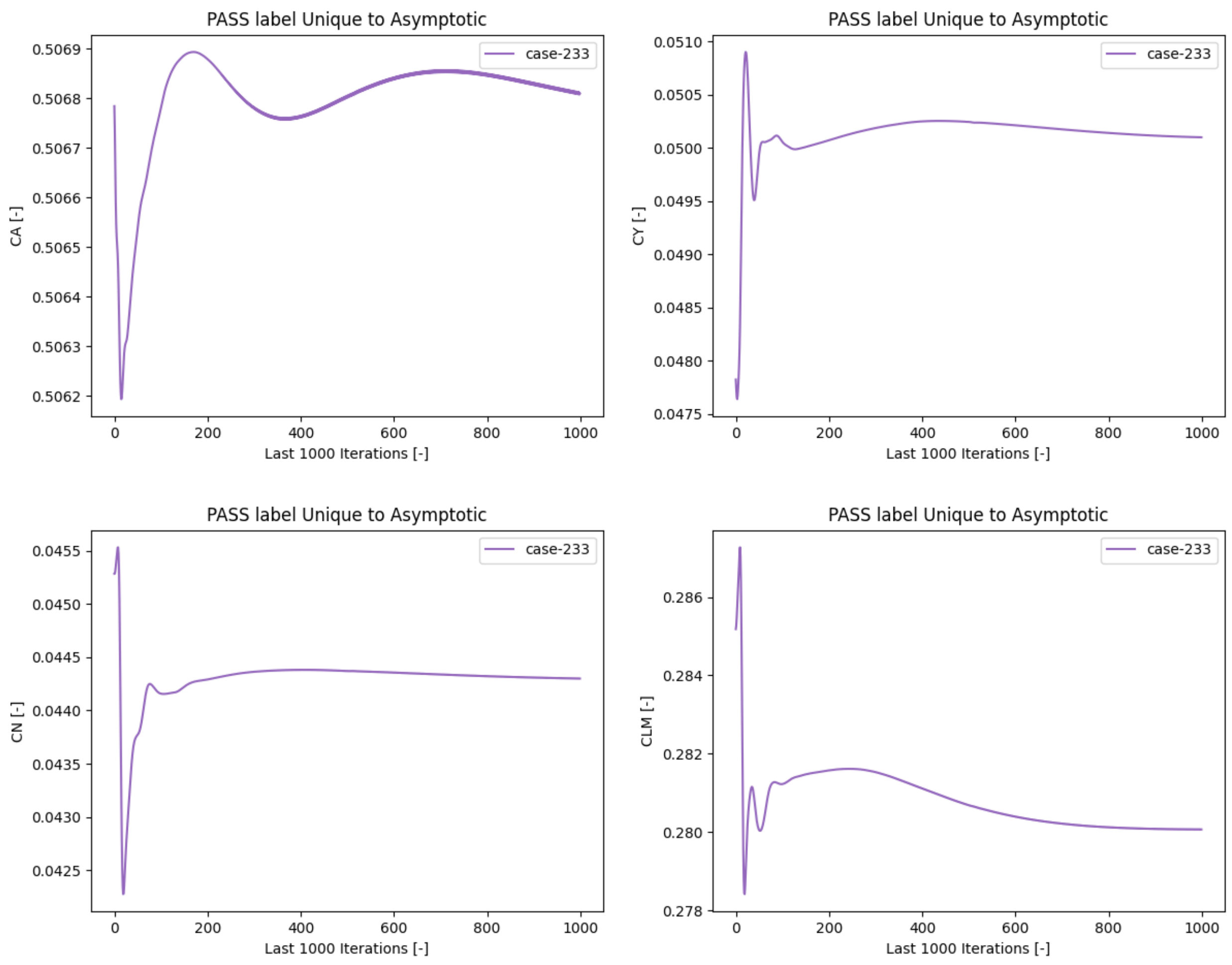


Figure 5.4. PASS labels unique to asymptotic without feature smoothing.

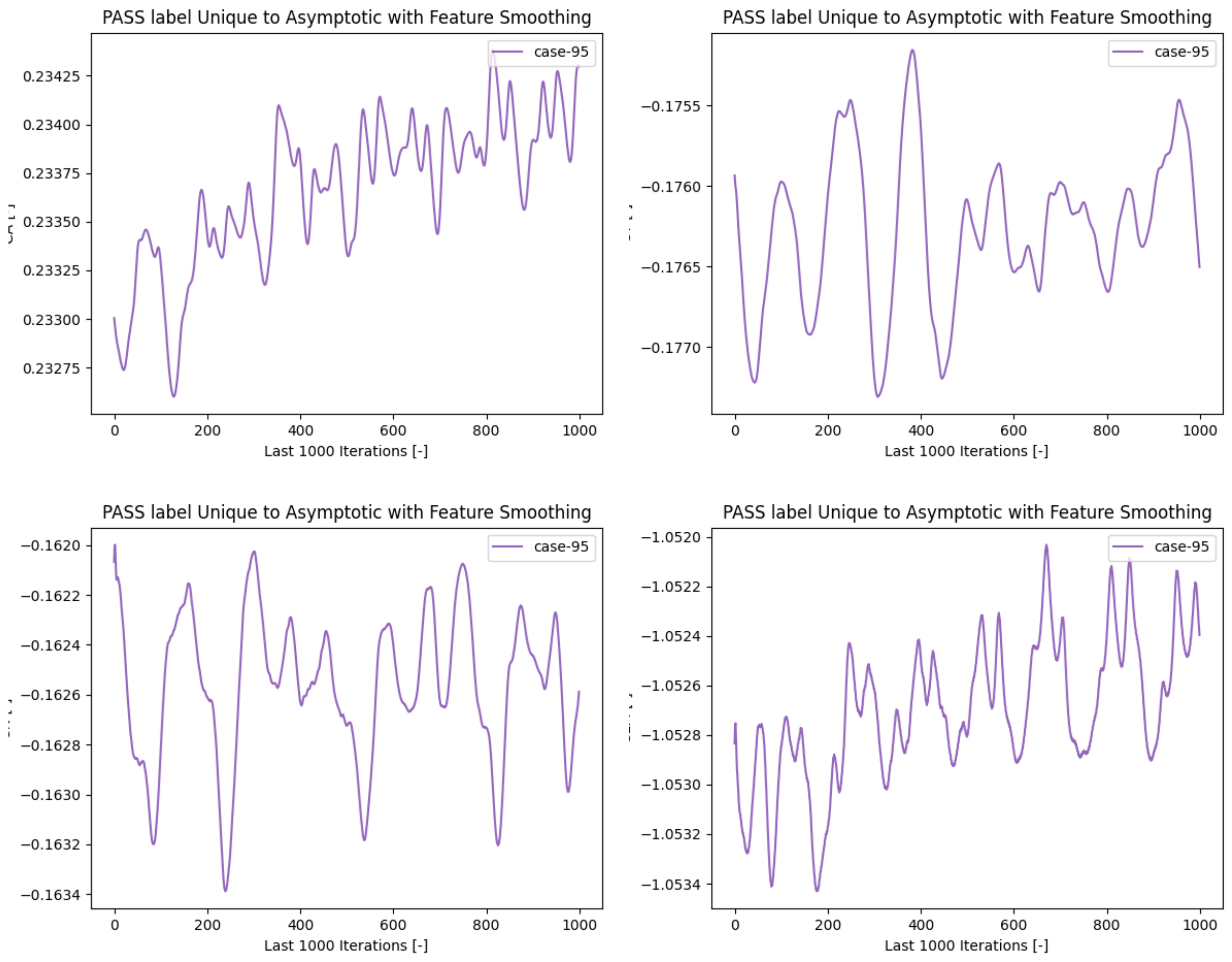


Figure 5.5. PASS labels unique to asymptotic with feature smoothing.

6. Results for Test Datasets

Prior to this chapter, all results presented pertained to the training and validations dataset discussed in section 5.0. These 300 cases were used to train and tune the machine learning modules. To test the performance of the final model, two datasets were constructed. The first dataset is comprised of 50 cases with Mach Numbers [2.6, 2,7, 2.8, 2.9, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5], total angles of attack [0.0, 4.0], and roll angles [0.0, 90, 180, 270]. The high Mach numbers associated with this dataset allow the simulations to converge quite well and quickly. The second dataset is comprised of 25 cases with Mach Numbers [0.5, 0.95, 1.10, 1.75, 2.5], angle of attacks [0.0, 4.0], and roll angles [0.0, 90, 180, 270]. These cases take considerably longer to converge, with the final few thousand iterations of Mach numbers 0.5, 0.95, and 1.10 being time averaged. Of these 25 cases, three cases experienced errors while running and are not included in the results.

6.0 High Supersonic Dataset Results

During the development of the code used to produce the results in this chapter, an unexpected occurrence was observed. When sampling batches of iterations from a case, a machine learning model may PASS a case only to then suggest extending the case in later samples. This means if only the last iterations of the case were used to pass judgement, then a converged case may be incorrectly labeled as EXTEND. Although more computationally expensive, stepping through batches of 1000 or 2000 iterations, 50 iterations at a time, proved successful for catching the “sweet spot” or regions that are more representative of the training and validation dataset).

Six cases were not classified as PASS by the machine learning model referenced in section 5.1. which considers a range of 1000 iterations. These cases were not debatable, and clear failures on the part of the machine learning model. To check if this was a cause of the iterations sampled, a second machine learning model which considered a range of 2000 iterations was trained and optimized. The new model which considered 2000 iterations only failed to PASS one case. Cases that were not passed by the machine learning model were given *NaN*’s and not presented within the results below. The absence of these cases is why the total number of iterations provided in the third column of table 6-1 for both machine learning models are given an asterisk.

Columns 4 through 7 of table 6-1 provide the average normalized delta between the coefficient at the last iteration of the window with which the case was classified PASS. Meaning if the asymptotic convergence criteria labeled the case as pass by looking at the iterative history range 3000-3500, then the coefficient used corresponds to iteration 3500. The user could judge if a case had converged every 500 iterations, while the automated methods could judge every 50 iterations starting at 2500. The difference between the convergence predictive method and the user was normalized by the difference between the maximum and minimum coefficient value from iteration 800 to the final iteration.

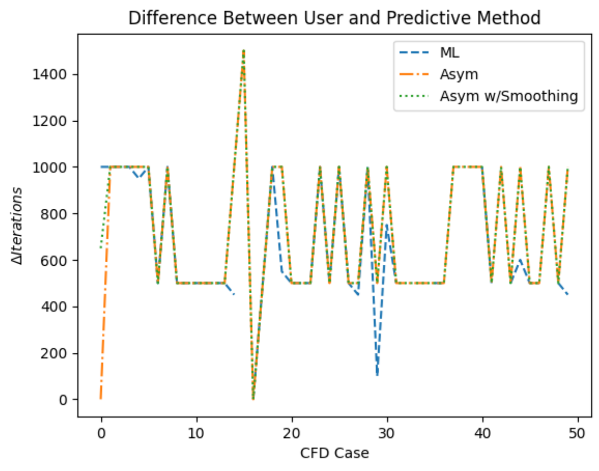
Table 6-1. Summary of each classification model's performance for high supersonic CFD cases.

	PASS	**EXTEND**	$\sum \boldsymbol{Iters}$	$\overline{\Delta CA}$ [%]	$\overline{\Delta CY}$ [%]	$\overline{\Delta CN}$ [%]	$\overline{\Delta CLM}$ [%]
User	50	0	161,500	-	-	-	-
ML Model – 1000 Iterations	44	6	110,200*	0.49	0.69	0.70	0.81
ML Model – 2000 Iterations	49	1	123,300*	0.50	0.75	0.71	0.77
Asymptotic	50	0	126,000	0.51	0.74	0.73	0.76
Asymptotic w/Smoothing	50	0	125,350	0.51	0.75	0.74	0.78

As seen in table 6-1, there is little to differentiate each classification method from another. Each model achieved a normalized difference in coefficient of less than 1%, and only differs from every other method a few hundredths of a percent. The main difference is the cases the machine learning models did not correctly label as PASS. Even if these cases had been included, as a collective, the models not only achieved solutions on average within 1% of the user but did so in roughly 23% fewer iterations.

Figures 6.1-6.5 depict the individual cases which makeup the values in Table 6-1. For figure 6.1, positive values in the difference between the number iterations required to label a case as PASS correlate to a lesser number than the user. Although the x-axis simply labels the cases 0 through 49, they are also ordered so that every five cases the Mach number increases. This is also true for the figures presented in section 6.1.

a)



b)

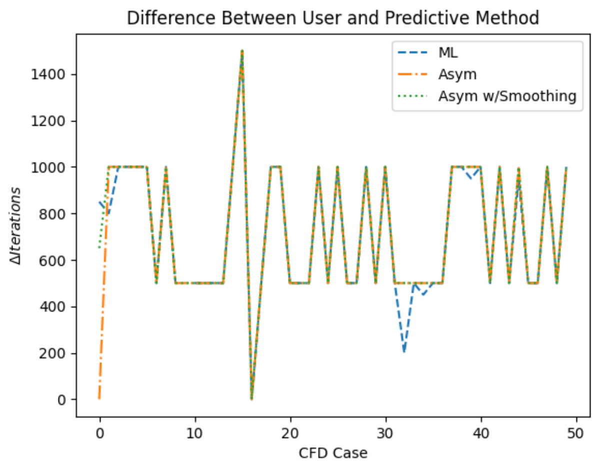
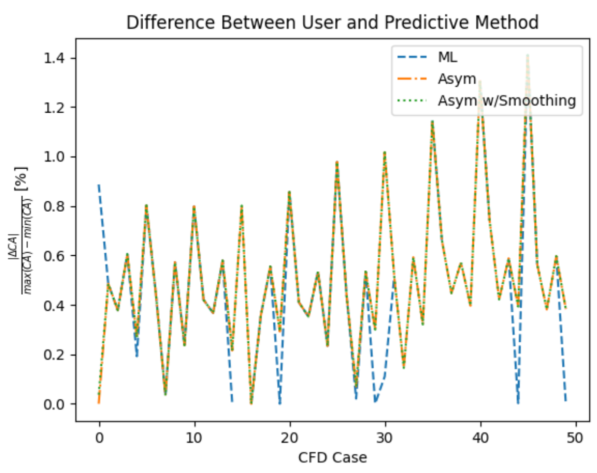


Figure 6.1. Difference in iterations taken to be classified as PASS.
a) 1000 iteration range ML model. b) 2000 iteration range ML model

a) b)



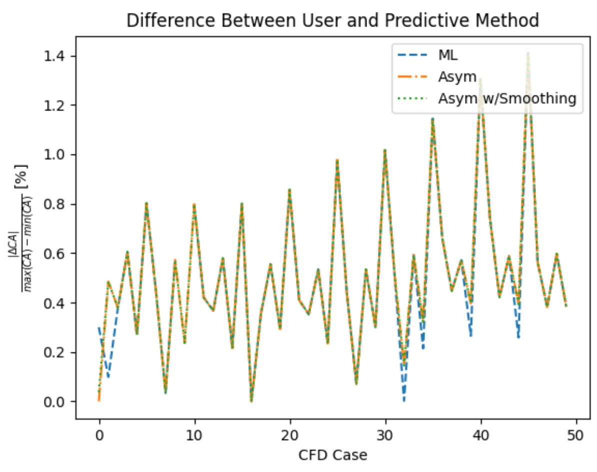
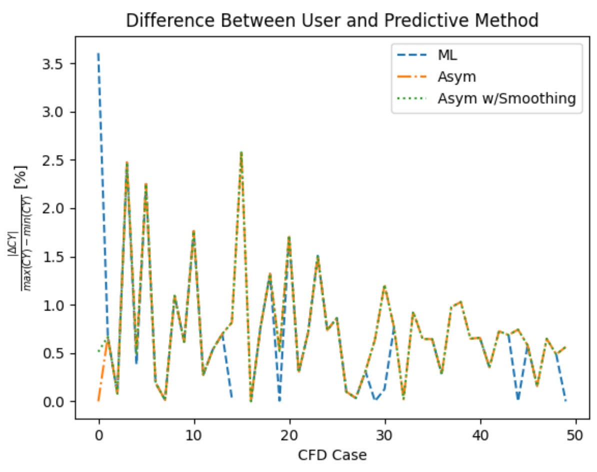


Figure 6.2. Difference in CA.
a) 1000 iteration range ML model. b) 2000 iteration range ML model

a) b)



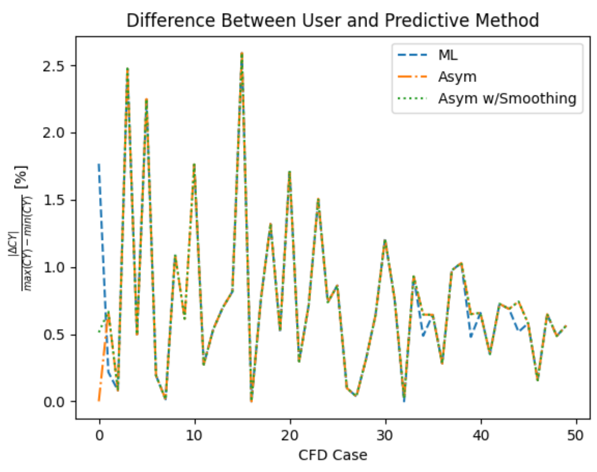
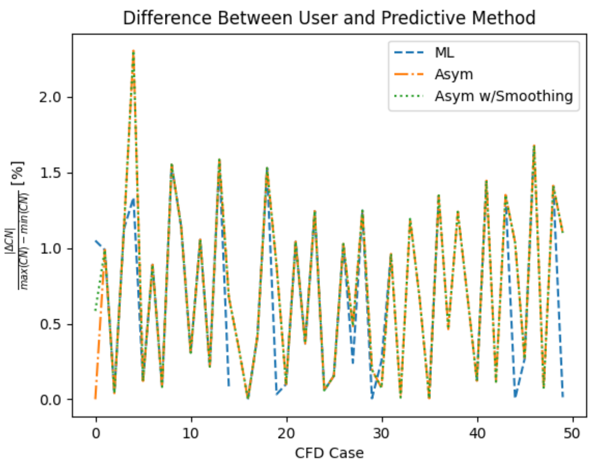


Figure 6.3. Difference in CY.
a) 1000 iteration range ML model. b) 2000 iteration range ML model

a)



b)

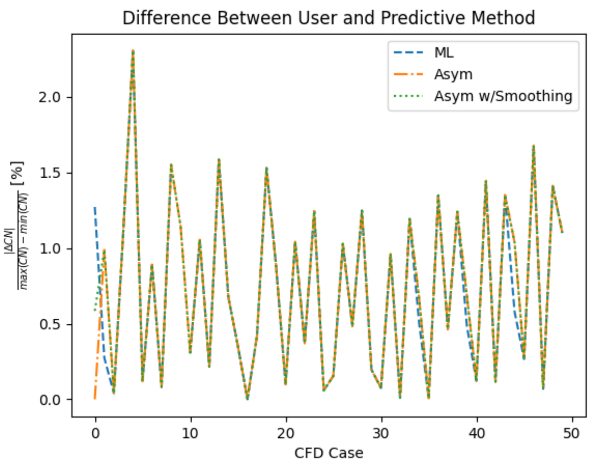
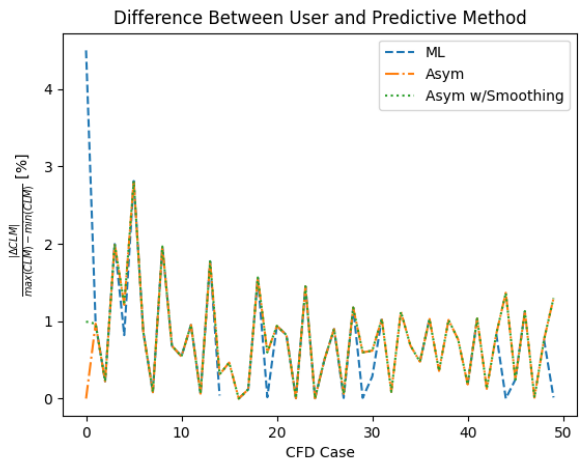


Figure 6.4. Difference in CN.
a) 1000 iteration range ML model. b) 2000 iteration range ML model

a)



b)

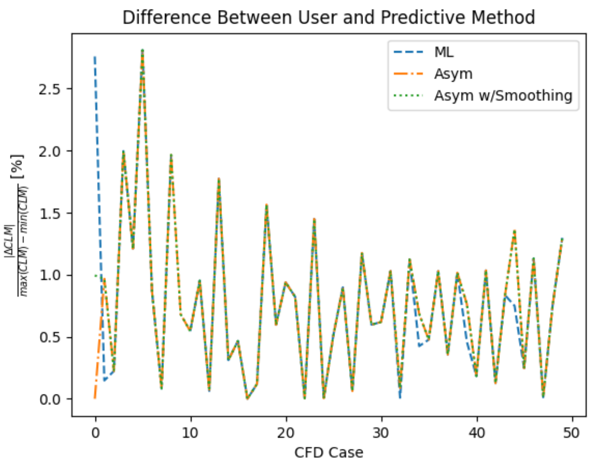


Figure 6.5. Difference in CLM.
a) 1000 iteration range ML model. b) 2000 iteration range ML model

6.1 Ascent Dataset Results

The ascent dataset was expected to challenge both the simple algorithms and machine learning models. Cases with a Mach number 1.10 or below can contain iterative histories displaying patterns indicative of transient flow features. Although half of the training and validation data consisted of simulations where the Mach number was 1.10 or below, only 38 of the 150 cases were labeled as PASS.

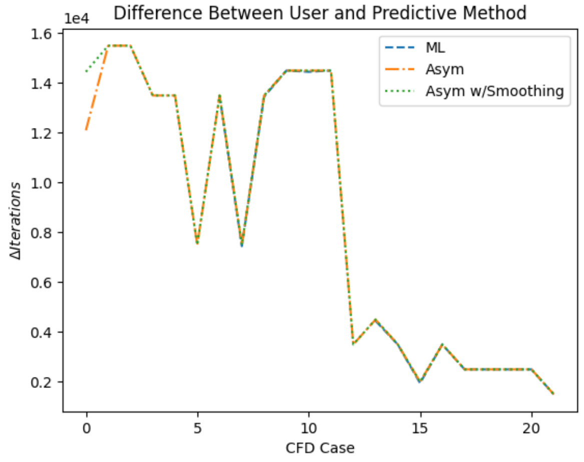
Table 6-2. Summary of each classification model's performance for ascent CFD cases.

	PASS	EXTEND	$\sum Iters$	$\overline{\Delta CA}$ [%]	$\overline{\Delta CY}$ [%]	$\overline{\Delta CN}$ [%]	$\overline{\Delta CLM}$ [%]
User	22	0	243,750	-	-	-	-
ML Model – 1000 Iterations	18	4	43,050*	2.21	4.28	4.57	3.81
ML Model – 2000 Iterations	21	1	60,550*	2.94	3.73	1.92	1.82
Asymptotic	22	0	59,650	3.41	3.42	4.31	3.26
Asymptotic w/Smoothing	22	0	57,300	3.38	3.43	4.33	3.25

As seen in table 6-2, the machine learning model which samples 2000 iterations achieved better coefficient deltas than the machine learning model which samples 1000 iterations and the samples algorithms. Like the high supersonic dataset, the machine learning model which samples 1000 iterations failed to classify four cases, and the machine learning model which samples 2000 iterations failed to classify one case. As a collective, the models decreased the total iterations by roughly 75%.

Figures 6.6-6.10 depict the individual cases which makeup the values in Table 6-2. Similar to section 6.0, positive values in the difference between the number iterations required to label a case as PASS correlate to a lesser number than the user. Additionally, every five cases the Mach number increases. Cases 0 to 14 represent the subsonic and transonic regime, and are responsible for the significant decrease in iterations, and the increase in deltas between coefficients.

a)



b)

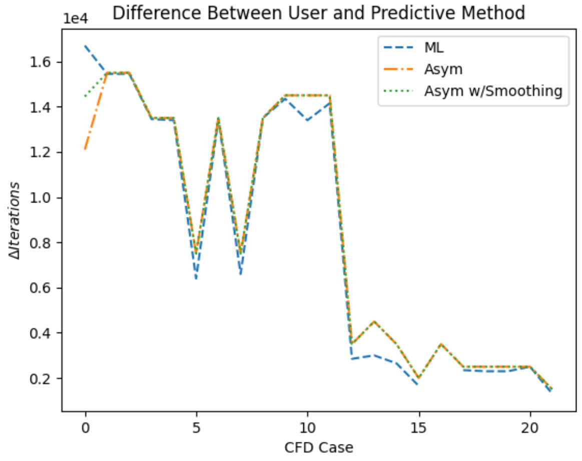
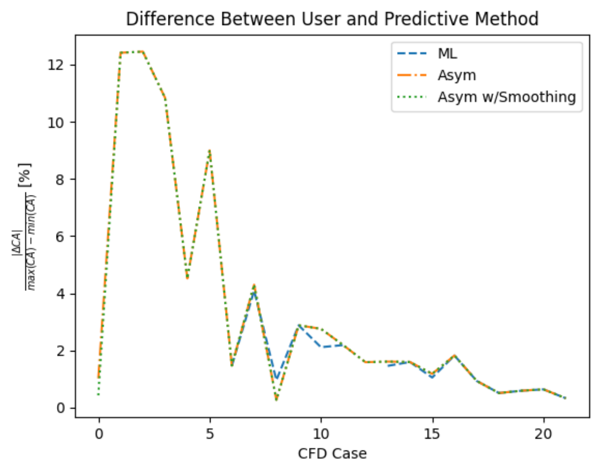


Figure 6.6. Difference in iterations taken to be classified as PASS.
a) 1000 iteration range ML model. b) 2000 iteration range ML model

a) b)



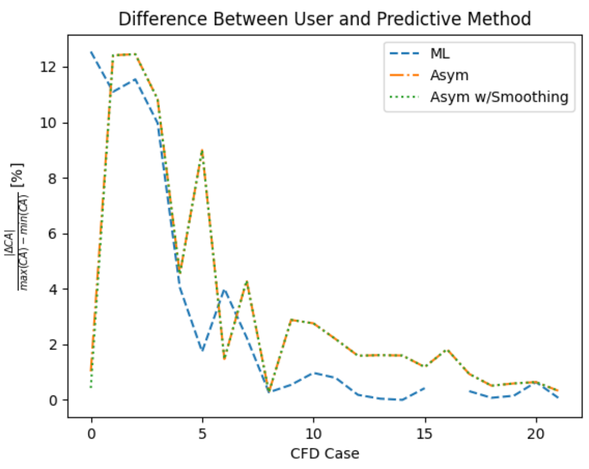
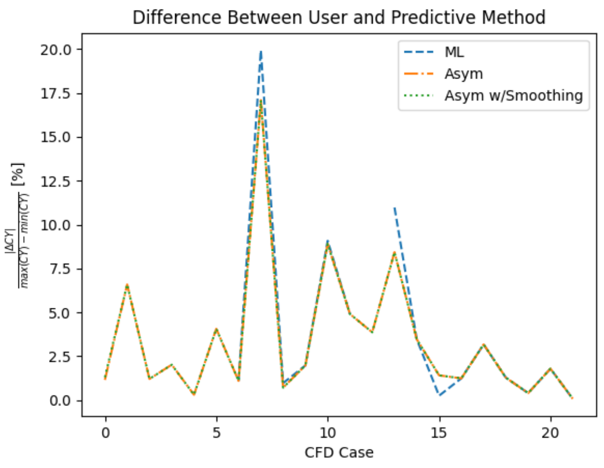


Figure 6.7. Difference in CA.

b) 1000 iteration range ML model. b) 2000 iteration range ML model

a) b)



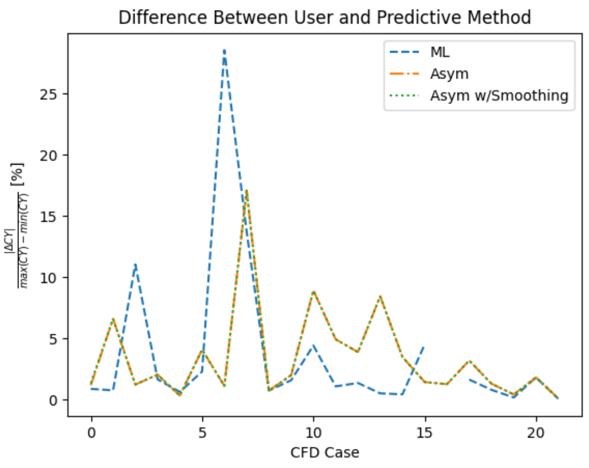
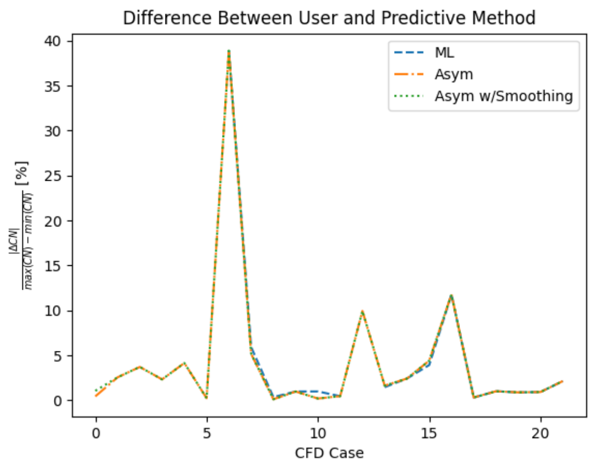


Figure 6.8. Difference in CY.

b) 1000 iteration range ML model. b) 2000 iteration range ML model

a) b)



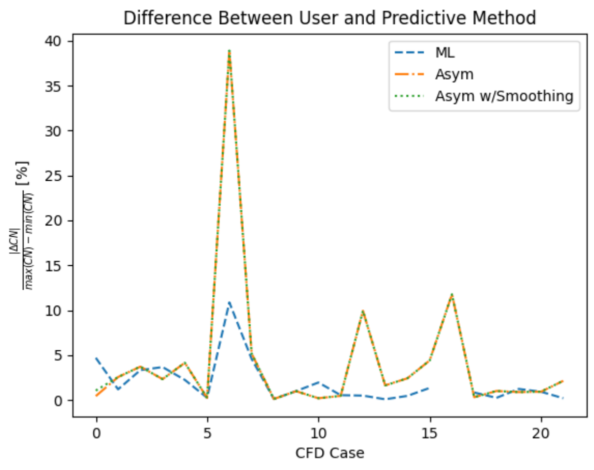
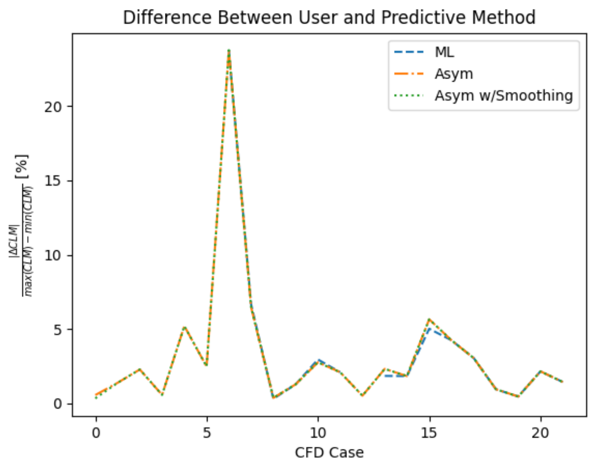


Figure 6.9. Difference in CN.
b) 1000 iteration range ML model. b) 2000 iteration range ML model

a) b)



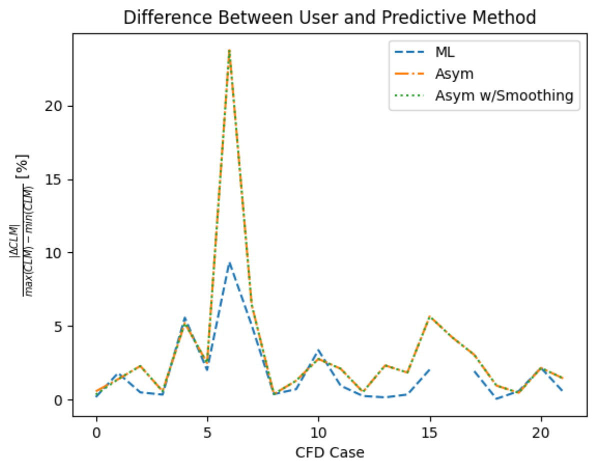


Figure 6.10. Difference in CLM.
a) 1000 iteration range ML model. b) 2000 iteration range ML model

7. Conclusion

Machine learning is often looked to as a tool for automating complex processes. In the field of computational fluid dynamics, properly classifying if a simulation has converged or not is one such problem. In this paper, two deep learning neural networks, trained on either 1000 or 2000 iterations per force coefficient, were compared to expert judgement and monitoring for asymptotic behavior with and without feature smoothing. A run matrix comprised of 50 supersonic cases and a run matrix comprised of 25 cases simulating an ascent trajectory were used to test each classification method. All methods achieved on average within 1% of the expert user's results for the supersonic run matrix and did so in fewer iterations for every case. For the ascent trajectory matrix, the machine learning model which sampled 2000 iterations achieved on average between 1.82-2.94%; a machine learning model which sampled 1000 iterations achieved between 2.21-4.57%; the asymptotic criteria achieved on average between 3.26-4.31%; and the asymptotic with smoothing criteria achieved on average between 3.25-4.33%. Each method had their largest differences for time accurate cases below Mach 1.10. For the machine learning models, this is mostly likely due to the disproportionate percent of PASS cases below Mach 1.10 in the training and validation dataset. One thing to note in the results is that the expert user's values were accepted as the baseline to compare each of the other methods. One of the purposes of pursing machine learning models is to reduce the inconsistences between practitioners. Comparing test case results for multiple expert users or running each test case for many more iterations could offer a more suitable baseline.

These results exemplify a distinction for implementing convergence criteria for steady and time accurate CFD cases. It is difficult to implement simple convergence methods across multiple regimes, whereas a unique machine learning model can be trained and utilized for a specific regime. Depending on the geometry of the flight vehicle being simulated, a steady solution may be satisfactory for certain cases in the subsonic and transonic regimes. A multi-class machine learning model could be capable of not only extending or passing cases but switching from a steady solver to time accurate. Additionally, more complex analyses such as a Fast Fourier Transform or Power Spectral Decomposition can be utilized to inform a machine learning framework on the number of iterations to consider.

References

[1] Valueva, M. V., Nagornov, N. N., Lyakhov, P. A., Valuev, G. V., and Chervyakov, N. I. "Application of the Residue Number System to Reduce Hardware Costs of the Convolutional Neural Network Implementation." *Mathematics and Computers in Simulation*, Vol. 177, 2020. https://doi.org/10.1016/j.matcom.2020.04.031.

[2] Is Your CFD Simulation Misbehaving? A Troubleshooting Checklist for Challenging Problems | Computational Fluid Dynamics (CFD) Blog – LEAP Australia & New Zealand. https://www.computationalfluiddynamics.com.au/cfd-troubleshooting-checklist/. Accessed Jun. 25, 2022.

[3] Rogers, S. E., Dalle, D. J., and Chan, W. M. "CFD Simulations of the Space Launch System Ascent Aerodynamics and Booster Separation." *AIAA*, Vol. 0778, 2015.

[4] Khalil, E. E. CFD History and Applications. *CFD Letters*. 2. Volume 4.

[5] Draelos, R. The History of Convolutional Neural Networks. *towardsdatascience.com*. https://towardsdatascience.com/a-short-history-of-convolutional-neural-networks-7032e241c483. Accessed Jun. 25, 2022.

[6] Li, X., Ao, Y., Guo, S., and Zhu, L. "Combining Regression Kriging With Machine Learning Mapping for Spatial Variable Estimation." *IEEE Geoscience and Remote Sensing Letters*, Vol. 17, No. 1, 2020, pp. 27–31. https://doi.org/10.1109/LGRS.2019.2914934.

[7] Wackers, J., Visonneau, M., Serani, A., Pellegrini, R., Broglia, R., and Diez, M. "Multi-Fidelity Machine Learning from Adaptive-and Multi-Grid RANS Simulations." 2020, pp. 18–23.

[8] Kochkov, D., Smith, J. A., Alieva, A., Wang, Q., Brenner, M. P., and Hoyer, S. "Machine Learning–Accelerated Computational Fluid Dynamics." *Proceedings of the National Academy of Sciences of the United States of America*, Vol. 118, No. 21, 2021.

[9] Zhao, Y., Akolekar, H. D., Weatheritt, J., Michelassi, V., and Sandberg, R. D. "RANS Turbulence Model Development Using CFD-Driven Machine Learning." *Journal of Computational Physics*, Vol. 411, 2020. https://doi.org/10.1016/j.jcp.2020.109413.

[10] Silva, V. B. R. E., and Cardoso, J. Chapter 3 - Overview of Biomass Combustion Modeling: Detailed Analysis and Case Study. In *Computational Fluid Dynamics Applied to Waste-to-Energy Processes* (V. B. R. E. Silva and J. Cardoso, eds.), Butterworth-Heinemann, 2020, pp. 87–121.

[11] Cummings, R. M., Mason, W. H., Morton, S. A., and McDaniel, D. R. *Applied Computational Aerodynamics: A Modern Engineering Approach*. Cambridge University Press, 2015.

[12] Obukhov, O., Smirnov, A., and Gysak, O. Numerical and Experimental Investigation of the Efficiency of Vaned Diffuser of Centrifugal Compressor. In *8th International Conference on Compressors and their Systems*, Woodhead Publishing, 2013, pp. 649–658.

[13] Hassan, Y. 12 - An Overview of Computational Fluid Dynamics and Nuclear Applications. In *Thermal-Hydraulics of Water Cooled Nuclear Reactors* (F. D'Auria, ed.), Woodhead Publishing, 2017, pp. 729–829.

[14] Martín~Abadi, Ashish~Agarwal, Paul~Barham, Eugene~Brevdo, Zhifeng~Chen, Craig~Citro, Greg~S.~Corrado, Andy~Davis, Jeffrey~Dean, Matthieu~Devin, Sanjay~Ghemawat, Ian~Goodfellow, Andrew~Harp, Geoffrey~Irving, Michael~Isard, Jia,

Y., Rafal~Jozefowicz, Lukasz~Kaiser, Manjunath~Kudlur, Josh~Levenberg, Dandelion~Mané, Rajat~Monga, Sherry~Moore, Derek~Murray, Chris~Olah, Mike~Schuster, Jonathon~Shlens, Benoit~Steiner, Ilya~Sutskever, Kunal~Talwar, Paul~Tucker, Vincent~Vanhoucke, Vijay~Vasudevan, Fernanda~Viégas, Oriol~Vinyals, Pete~Warden, Martin~Wattenberg, Martin~Wicke, Yuan~Yu, and Xiaoqiang~Zheng. {TensorFlow}: Large-Scale Machine Learning on Heterogeneous Systems.

[15] Bisong, E. TensorFlow 2.0 and Keras. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*, Apress, Berkeley, CA, 2019, pp. 347–399.

[16] Ertam, F., and Aydın, G. Data Classification with Deep Learning Using Tensorflow. 2017.

[17] Abu, M. A., Indra, N. H., Rahman, A. H. A., Sapiee, N. A., and Ahmad, I. "A Study on Image Classification Based on Deep Learning and Tensorflow." *International Journal of Engineering Research and Technology*, Vol. 12, No. 4, 2019, pp. 563–569.

[18] Do, Q., Son, T. C., and Chaudri, J. "Classification of Asthma Severity and Medication Using TensorFlow and Multilevel Databases." *Procedia Computer Science*, Vol. 113, 2017, pp. 344–351. https://doi.org/https://doi.org/10.1016/j.procs.2017.08.343.

[19] O'Malley, T., Bursztein, E., Long, J., Chollet, F., Jin, H., Invernizzi, L., and others. KerasTuner.

Appendix A – Model Script

train-optimize-multifeature-fun.py

```
"""
This script imports features and label data, preps the data for
use, builds the model architecture, and optimizes the
hyperparameters of said model. These tasks are completed with
the help of three modules:
1) importdata.py
2) prepdata.py
3) buildmodel*.py
"""

# Standard library modules
import os
import sys
import time

# TensorFlow modules
from tensorflow import keras
from keras_tuner import tuners
from keras.callbacks import TensorBoard

# Local modules
from modules import importdata
from modules import prepdata
from modules import buildmodel_binary_funAPI as bm

# Add module folder to system path
module_path =
os.path.join(os.path.dirname(os.path.abspath(__file__)),'modules
')
print(module_path)
sys.path.insert(0, module_path)
```

```
# Track run time
startTime = time.time()
time_tracker = int(time.time())

# Training method
train_method = "import" # options ["import", "split"]

# Define Basic Model Parameters
iters_conv = 1000 # Iteration count to consider
iters_start = 800 # Iterations to start from for EXTEND case
asym_threshold = 0.0015 # Max difference between values for
classification
iters_asym = 50 # Number of iterations to consider for
asymptotic

epochs_num = 10 # Number of training iterations
comb_num = 1 # Number of hyperparameter combinations
init_rand = 1 # Number of random searches prior to Bayesian Opt
train_vs_test = 0.80 # Percent of data devoted to training

# Define up one of file directory
current_path =
os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
# Define where data lives
feature_path = 'data/sls/run/10000/f3_dac0'
# Label path
label_path = 'survey/results/clv-survey-results-ddalle.csv'
# Files to load. Change the function 'update_graph' to match
files_coeff = ['CA','CY','CN','CLM'] # number of branches
dependent on coeff

# Load coefficient data
```

```
coeffs, cases_num = importdata.features_mat(current_path,
feature_path, files_coeff)


if train_method == "split":
    '''Train based on splitting the iterative history'''
    # Convert features and labels into TF dataset
    dataset_train, dataset_test = prepdata.split(coeffs,
cases_num, iters_start, iters_conv, train_vs_test)
elif train_method == "import":
    '''Train based on surveyed labels'''
    # Import labels
    labels =  importdata.labels_csv(current_path, label_path)
    # Import features
    features = prepdata.features(coeffs, cases_num, iters_conv)
    # Convert features and labels into TF dataset
    dataset_train, dataset_test = prepdata.dataset(features,
labels,  train_vs_test)
else:
    print(
        'Invalid section. Choose either "import" or "split"'
    )
    raise

# Name of tensorboard log folder
NAME = "TF-FunAPI-{}FV-{}iters-{}-{}comb-
{}".format((len(files_coeff)),iters_conv,train_method, comb_num,
time_tracker)

# For tracking and visualization
tb_path = "logs/{}".format(NAME)
tb = TensorBoard(log_dir=tb_path)
```

```
my_hyper_model = bm.MyHyperModel(iters_conv=iters_conv,
files_coeff=files_coeff)
# BayesianOptimization - probabilistic approach that takes into
account already tested combinations to sample the next
combination
tuner = tuners.bayesian.BayesianOptimization(
    hypermodel=my_hyper_model, # model-building function
    objective='val_loss', # diff btwn true and predicted values
    num_initial_points=init_rand, # Randomly generated samples
for training
    max_trials=comb_num, # number of combinations to test
    executions_per_trial=1, # number of times to run each
combination
    directory=tb_path, # where to store optimization history
)

# Equivalent to model.fit
tuner.search(
    dataset_train, # training dataset
    epochs = epochs_num, # training iterations
    validation_data = dataset_test, # test dataset
    callbacks=[tb], # TensorBoard callback to visualize results
    verbose=2, # verbose=2 is recommended when not running
interactively
)

# Print results and save best model
print(tuner.results_summary()) # displays top 10
print(tuner.get_best_models()[0].summary()) # displays best
model
model = tuner.get_best_models(num_models=1)[0] # get best model

# Save best performing model
model.save(
    os.path.join(
        current_path,
        "models/optimized/",
        NAME
    ),
    overwrite=True,
)

# Save figure of model. Requirements: brew install
[svn/graphviz]
keras.utils.plot_model(
    model,
    os.path.join(
        current_path,
        "code/figs/{}.png".format(NAME)
        ),
    show_shapes=True
)

#Run Time
executionTime = (time.time() - startTime)
print('Execution time in seconds: ' + str(executionTime))
```

Appendix B – Model Modules

importdata.py

```
import os
import json
import scipy.io as sio
import pandas as pd
import numpy as np
from collections import defaultdict


def features_mat(current_path, file_path, files_coeff):
    """
    Import each coefficient's associated mat file. Compile into
    a list of dictionaries. [iterations, case] addition of nan
    since varying number of iterations.
    """
    file_ext = '.mat' # currently must be mat files
    # Load Data as list of dictionaries
    # [iterations, case] addition of nan since varying number of
    iterations
    coeffs = []
    for each in files_coeff:
        # loads each force coefficient as a dictionary
        coeffs.append(sio.loadmat(os.path.join(current_path,
file_path, each + file_ext)))
        # Determine sample size
        cases_num = len(coeffs[0][files_coeff[0]][0,:])
    return coeffs, cases_num

def labels_csv(current_path, label_path):
    # Initiate object
    labels = []
    # Combine path
    path = os.path.join(current_path, label_path)
    # Read in csv with labels
    labels_csv = pd.read_csv(path)
    for row in labels_csv['PASS (1 == PASS, 0 == EXTEND)']:
        labels.append(row)
    return labels


# Full genericity
def makehash():
    return defaultdict(makehash)


def trials_json(current_path, metrics, trials):
    """
    Read each trial json and assign specified metrics to a
    nested dictionary.
    """

    # Grab metrics and save as nested dict
    trial_dict = makehash()

    for trial in trials:
        # Count number of trial
        totalDir = 0
        for base, dirs, files in
os.walk(os.path.join(current_path,'logs/{}/untitled_project'.for
mat(trial))):
            # print('Searching in : ',base)
            for directories in dirs:
                totalDir += 1
        dir_range = np.arange(0,totalDir,1)
        for dir in dir_range:
            # Define path to optimization logs
            if dir < 10:
                json_path =
'logs/{}/untitled_project/trial_0{}'.format(trial,dir)
            else:
                json_path =
'logs/{}/untitled_project/trial_{}'.format(trial,dir)
            # json file name
            json_file = 'trial.json'
            # open json file
            f =
open(os.path.join(current_path,json_path,json_file))
            # load json
            data = json.load(f)
            for metric in metrics:
                trial_dict[trial][dir][metric] =
data["metrics"]["metrics"][metric]["observations"][0]["value"]

    return trial_dict
```

prepdata.py

```
from math import nan, isnan
import numpy as np
import tensorflow as tf

def features(coeffs, cases_num, iters_conv, iters_consider =
None):
    """
    Removes nan from each case's coefficients. Take last group
    of specified iterations. Returns a list of feature variables
    and their features.
    """
    # Initialize data set inputs
    features = []
    # For each coefficient
    for count, coeff in enumerate(coeffs):
        # Get dict keys
        coeff_name = []
        for key in coeff.keys():
            coeff_name.append(key)

        # Initiate feature array
        feature = np.zeros((cases_num,iters_conv,len(coeffs)))

        # For each case, remove nan's and save coefficient
history
        for colm in range(len(coeff[coeff_name[3]][0,:])):
            # Remove nan per case
            coeff_colm =
coeff[coeff_name[3]][:,colm][~np.isnan(coeff[coeff_name[3]][:,co
lm])]
            # Save features and associated label
            if iters_consider == None:
                feature[colm,:,count] = coeff_colm[
                    -iters_conv : iters_consider
                    ]
            else:
                feature[colm,:,count] = coeff_colm[
                    iters_consider-iters_conv : iters_consider
                    ]

        features.append(feature[:,:,count])

    return features


def dataset(features, labels, train_vs_test):
    """
    Combines prepped features and imported labels into two
    tensorflow datasets. One for training and one for
    validation.
    """

    # Number of cases
    cases_num = np.size(features[0][:,0])

    # tf.data.Dataset API supports efficient input pipelines
    dataset = tf.data.Dataset.from_tensor_slices(
        (
            tuple(features),
            labels
        )
    )
    # Shuffle data
    dataset = dataset.shuffle(10000)

    # Allocate portion of data to either training or testing
    train_size = int(cases_num * train_vs_test)
    # Everything up to train_size
    dataset_train = dataset.take(train_size).batch(20)
    # Everything after train_size
    dataset_test = dataset.skip(train_size).batch(10)

    return dataset_train, dataset_test


def split(coeffs, cases_num, iters_start, iters_conv,
train_vs_test):
    """
    Remove nan from each case's coefficients. Split coefficient
    history into extend and pass cases based on beginning and
    end iterations. Returns features and associated labels as
    tensorflow datasets.
    """
    # Initialize data set inputs
    labels = []
    features = []
    # For each coefficient
    for count, each in enumerate(coeffs):
        # Get dict keys
        coeff_name = []
        for key in each.keys():
            coeff_name.append(key)

        # Initiate feature array
        feature = np.zeros((2*cases_num,iters_conv,len(coeffs)))

        # For each case, remove nan's and save coefficient
history
        for colm in range(len(each[coeff_name[3]][0,:])):
            # Remove nan per case
            coeff_colm =
each[coeff_name[3]][:,colm][~np.isnan(each[coeff_name[3]][:,colm
])]
            # Save features and associated label
            feature[colm,:,count] = coeff_colm[-iters_conv:]
            # Assign label
            if count == 0:
                labels.append(1)
            else:
                pass

        # delete this section
        for colm in range(len(each[coeff_name[3]][0,:])):
            # Remove nan per case
            coeff_colm =
each[coeff_name[3]][:,colm][~np.isnan(each[coeff_name[3]][:,colm
])]
            feature[cases_num+colm,:,count] =
coeff_colm[iters_start:iters_start+iters_conv]
            # Assign label
            if count == 0:
                labels.append(0)
            else:
                pass

        features.append(feature[:,:,count])

    # tf.data.Dataset API supports efficient input pipelines
    dataset = tf.data.Dataset.from_tensor_slices(
        (
            tuple(features),
            labels
        )
    )
    # Shuffle data
    dataset = dataset.shuffle(10000)

    # Allocate portion of data to either training or testing
    train_size = int(2*cases_num * train_vs_test)
    # Everything up to train_size
    dataset_train = dataset.take(train_size).batch(20)
    # Everything after train_size
    dataset_test = dataset.skip(train_size).batch(10)

    # Return tensorflow datasets
    return dataset_train, dataset_test


def smooth_features(files_coeff, cases_num, features):
    '''
    Account for oscillations in the coefficient iterative
    history by smoothing the features. Walks through features
    and averages across the interval range. Returns identical
    formatting as prepdata.features.
    '''

    # Initialize list
    features_smoothed = []
    for count,coeff in enumerate(files_coeff):
        # for coefficients
        holder = []
        for case in range(cases_num):
            # for number of cases
            x_del = 0
            avg_np = []
            for scan in range(100):
                # how many steps to walk
                x = -np.size(features[count][case,:])
                x = x + x_del
                average = []
                for interval in range(20):
                    y = x + 100
                    if y == 0 and x_del == 0:
                        tmp =
np.average(features[count][case,x:])
                    elif x > -100:
                        tmp = np.nan
                    else:
                        tmp =
np.average(features[count][case,x:y])
                    x += 100

                    average.append(tmp)

                avg = np.array(average)
                avg_np.append(avg)
                x_del += 1
            holder.append(avg_np)

        coeff_smooth = []
        for case in range(cases_num):
            a = []
            for each in range(np.size(holder[0][0])):
                for every in range(9):
                    a.append(holder[case][every][each])
            coeff_smooth.append(a)
        features_smoothed.append(coeff_smooth)

    for count,coeff in enumerate(files_coeff):
        for case in range(cases_num):
            features_smoothed[count][case] = [x for x in
features_smoothed[count][case] if isnan(x) == False]
```

```
features_tmp = []
for count,coeff in enumerate(files_coeff):
    features_tmp.append(np.array(features_smoothed[count]))
features_smoothed = features_tmp

return features_smoothed
```

predict.py

```
"""
This script is capable of two prediction types.
    1) Using a pre-trained machine learning model.
    2) By checking for asymptotic behavior
"""

import os
import numpy as np
import tensorflow as tf

def labels_model(model_name, features):
    '''
    Make prediction based on a pre-trained TensorFlow model.
    Returns list of predictions.
    '''
    # Define up one of file directory
    current_path =
os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(
__file__))))

    # Load tensorflow model
    model = tf.keras.models.load_model(
        os.path.join(
            current_path,
            "models/optimized/{}"
            ).format(model_name)
    )
    # Continue training
    prediction = np.round(
        model.predict(
            features, # input sample
        )
    )

    return prediction



def labels_asymptotic(files_coeff, cases_num, features,
asym_threshold,iters_asym):
    '''
    Determine if the difference between the maximum and minimum
    coefficient values in the last specified iterations is below
    the asym_threshold argument. Return a list of labels based
    on the criteria
    '''

    # Number of coefficients
    coeff_num = np.size(files_coeff)

    # Initialize numpy array
    coeff_diff = np.zeros([cases_num,coeff_num])
    # For each coefficient
    for count,coeff in enumerate(files_coeff):
        # For each case
        for i in np.arange(cases_num):
            # max value
            coeff_max = np.max(features[count][i,-iters_asym:])
            # min value
            coeff_min = np.min(features[count][i,-iters_asym:])
            # difference between max and min
            coeff_diff[i, count] = (np.abs(coeff_max-coeff_min))

    # Initialize list
    diff_max = []
    # For each case
```

```
for i in np.arange(cases_num):
    # Max value between coefficients
    diff = np.max(coeff_diff[i,:])
    diff_max.append(diff)

# Initialize list
labels = []
# For each case
for i in np.arange(cases_num):
    # Assign PASS if below threshold
    if diff_max[i] <= asym_threshold:
        labels.append(1)
    # Assign EXTEND otherwise
    else:
        labels.append(0)

# Return list of labels
return labels
```

buildmodel_binary_funAPI.py

```
import tensorflow as tf
from tensorflow import keras
from kerastuner.engine.hyperparameters import HyperParameters as
hp
from kerastuner import HyperModel

class MyHyperModel(HyperModel):

    def __init__(self, iters_conv, files_coeff):
        # Additional model arguments
        self.iters_conv = iters_conv
        self.files_coeff = files_coeff

    def build(self, hp):
        ''' Constructs the model's architecture. Includes a
        branch for each coefficient consisting of an input
        layer, hidden layers, and an output layer. Ends with a
        trunk which merges the branches and consist of an input
        layer, hidden layers, and an output.'''

        # hyperparameters
        # input layer density
        input_density = hp.Int(
            "input_density",
            min_value=128,
            max_value=192,
            step=32,
        )
        # number of hidden layers in each branch
        n_layers_branch = hp.Int("n_layers_branch", 3,3)
        # input layer density when merging branches
        input_merge_density = hp.Int(
            "input_merge_density",
            min_value=18,
            max_value=36,
            step=6,
        )
        # number of hidden layers after merge
        n_layers_merge = hp.Int("n_layers_merge",3,3)

        # Initialize layer input and output holders
        coeff_output = []
        coeff_input = []
        for coeff in self.files_coeff:
            coeff = coeff.lower()
            # define two sets of inputs
            input_coeff = keras.Input(shape=(self.iters_conv,),
name=coeff)

            # Define input layers
            dense_input_coeff = keras.layers.Dense(
                input_density,
                activation="relu",
                name='Branch-Hidden_Layer-0-{}'.format(coeff)
            )

            # Branch for each input
            locals()[coeff] = dense_input_coeff(input_coeff)
            for i in range(n_layers_branch):
                # Define Hidden Layer
                dense_hidden = keras.layers.Dense(
                    hp.Int(
                        "dense_{}_units_branch".format(i),
                        min_value=8,
                        max_value=64,
                        step=8,
                    ),
                    activation="relu",
                    name='Branch-Hidden_Layer-{}-
{}'.format(i+1,coeff)
                )
                locals()[coeff] = dense_hidden(locals()[coeff])
            locals()[coeff] = keras.Model(
                inputs=input_coeff,
                outputs=locals()[coeff]
            )
            coeff_output.append(locals()[coeff].output)
            coeff_input.append(locals()[coeff].input)

        # combine the output of the two branches
        combined = keras.layers.concatenate(
            coeff_output
        )

        # combined layers
        # Define Hidden Layer
        dense_merge = keras.layers.Dense(
            input_merge_density,
            activation="relu",
            name='Trunk_Hidden_Layer-0'
        )
        m = dense_merge(combined)

        for i in range(n_layers_merge):
            # Define Hidden Layer
            dense_merge_hidden = keras.layers.Dense(
                hp.Int(
                    "dense_{}_units_merge".format(i),
                    min_value=9,
                    max_value=21,
                    step=3,
                ),
            activation="relu",
            name='Trunk_Hidden_Layer-{}'.format(i+1)
            )
            m = dense_merge_hidden(m)


        m = keras.layers.Dense(1, activation="sigmoid",
name='Trunk-Output_Layer-Sigmoid')(m)

        # then output a single value
        model = keras.Model(
            inputs=coeff_input,
            outputs=m,
            name="Multi-Feature_Variable-Binary_Classification-
Functional_API"
        )

        # Define optimizer
        opt = keras.optimizers.Adam(hp.Choice("learning_rate",
values=[1e-2,1e-3,1e-4]))
        # Configure model for training
        model.compile(
            # True if no activation function is used in final
layer
            # True may be more numerically stable. Applies own
sigmoid transformation

loss=keras.losses.BinaryCrossentropy(from_logits=False),
            optimizer=opt,
            metrics=[tf.keras.metrics.BinaryAccuracy()]
        )
```

```
    return model
```