ARTIFICAL NEURAL NETWORK HYPERPARAMETER EFFECTIVENESS DETERMINATION AND OPTIMIZATION ALGORITHM

James VanderVeen

AER870 Aerospace Engineering Thesis – Final Report

Faculty Advisor: Dr. Krishna D. Kumar

Date: April 2021

Acknowledgements

The author would like to acknowledge the following people. Dr. Kumar, whose guidance and patience enable me to complete this thesis. My family for their infinite love and support throughout my life. And my partner, Chantel, for the love, support, encouragement I couldn't do without.

Abstract

Machine learning models can contain many layers and branches. Each branch and layer, contain individual variables, know as hyperparameters, that require manual tuning. For instance, the genetic algorithm designed by Unit Amin [2] was designed to mimic the reproductive process of living organisms. The genetic algorithm and the Artificial Neural Network (ANN) training processes contain inherent randomness that reduces the replicability of results. Combined with the sheer magnitude of hyperparameter permutations, confidence that model has arrived at the best solution may be low. The algorithm designed for this thesis was designed to isolate portions of a complex ANN model and generate results showing the effect each hyperparameter has on the performance of the model as a whole. The results of this thesis show that the algorithm effectively generates data correlating model performance to hyperparameter selection. This is evident in section 3.1, and 3.2, where it is shown that using the *sigmoid* activation function with CNN layers, regardless of the number of filters, or hyperparameters used in the subsequent LSTM layers, produces superior RMSE scores. Section 3.2 also reveals that the model does not improve in performance as the number of CNN and LSTM layers are added to the model. Finally, the results in section 3.3 show that the *rmsprop* optimizer generates superior results regardless of the hyperparameters used in the rest of the model.

Table of Contents

Nomenclature

ANN	Artificial Neural Network
AHP	AIPAAS Hybrid Project
CMB	Condition Based Maintenance
CMAPSS	Commercial Modular Aero-Propulsion System Simulation
CNN	Convolutional Neural Network
HPC	High-Pressure Compressor
HPT	High-Pressure Turbine
HCL	Hybrid CNN/LSTM
LPC	Low-Pressure Compressor
LPT	Low-Pressure Turbine
LSTM	Long Short-Term Memory
PHM	Prognostics and Health Management
RMSE	Root Mean Square Error
RUL	Remaining Useful Life

List of Figures

List of Tables

1 Introduction

The original goal of this thesis was to design a novel model to predict the Remaining Useful Life (RUL) of a turbojet engine using simulation data generated by the National Aeronautics and Space Administration's (NASA) Commercial Modular Aero-Propulsion Simulation System (CMAPSS) [1]. In preparation for the design, the hybrid Convolutional Neural Network (CNN)/ Long Short-Term Memory (LSTM) neural network developed by Amin [2], for his master's thesis, was utilized to learn how to build and train neural networks. During that training period, the author was advised to investigate the discrepancies in the results generated by the Hybrid CNN-LSTM (HCL) model.

Models	Scores	FD001	FD002	FD003	FD004	PHM08
HDNN	RMSE	13.017	15.24	12.22	18.156	-
	s-score	-	-	-	-	-
DCNN	RMSE	12.61	22.36	12.64	23.31	-
	s-score	-	-	-	-	-
Deep LSTM	RMSE	16.14	24.49	16.18	28.17	-
	s-score	338	4450	852	5550	1862
MODBNE	RMSE	15.04	25.05	12.51	28.66	-
	s-score	334	5585	421	6557	-
BiLSTM-ED	RMSE	14.74	22.07	17.48	23.49	-
	s-score	273	3099	574	3202	1098
Growing RNN	RMSE	14.31	23.71	16.42	27.95	-
	s-score	302.80	4105.56	457.49	4906.04	1356.13
Hybrid models used in project	RMSE	15.68	22.26	16.89	26.32	-
	s-score	-	-	-	-	-

Figure 1: Comparison of APH to other models [2].

Figure 4 shows the results of the HCL model compared to results of other models from Amin's literature review. What stands out is that the HCL model didn't consistently perform better or worse than the other models across all four datasets. In ranking the HCL's performance it can be seen that the best relative results come from the model trained using the FD002 dataset, where it achieved the 3rd lowest Root Mean Square Error (RMSE). Whereas the FD001, FD003, and FD004 results are ranked 6th, 6th, and 4th, respectively. From these observations it was decided that instead of blindly learning how to build and train neural networks, it would be wiser to also learn about the inner workings of the model.

1.1 Alternate PHM Approaches

It is never a good idea to follow a path, without knowing why that path is the correct choice. With that in mind, alternate methods of determining the RUL of a system were also studied. According to Goodman et al. [3], there are three main approaches to PHM, classical, usage-

based, and condition-based. Classical based methods are further broken down into three categories, model-based, data-driven, or hybrid-driven [3].

1.1.1 Model-Based PHM

Model-based PHM is the most familiar for an undergraduate student as it uses physics, and reliability statistics to determine the RUL of a system. Determining the yield point of a steel bar is a good example. Data-driven models use previous generated data, and statistics to predict the RUL of a system. Data-driven models attempt to derive probabilities of future events from a history of a past event chains. For example, if the weather was 7°C and cloudy on Monday, what is the likelihood of rain on Tuesday? A hybrid system combines model-based and data-driven to optimize a PHM system for precision and ease-of-development [3]. Model-based designs are highly accurate because physics and statistics are inherently accurate. Model-based designs lose their abilities as systems become more complex, not because the underlying principle are wrong, but because it becomes harder to determine the conditions to which each component is subjected. Whereas a data-driven model only gives answers for the conditions in which it was generated. A hybrid approach then takes the strengths of each model type, achieving greater performance than either alone [3].

1.1.2 Usage-Based PHM

Usage-based models are those that monitor a system and make predictions on the future health of a system by accounting for the past and present usage history and the physics-based model of the system [4]. This is basically a hybrid model that updates the statistical models as the usage history is gained. The need for this type of model arises from the weakness in physical models, the uncertainty in operating conditions. Furthermore, the uncertainty in a future event grows the further into the future a prediction is made. This can be demonstrated by an object travelling in a straight line, whose intent is to travel 1 meter forward. If it is known that the objects path may deviate, from directly forward, by one centimeter after one meter of forward travel, then after 10 timesteps the actual location could be between 10 centimeters to the left and 10 centimetres to the right. A usage-based model would update itself at each timestep, such that the model could predict the outcome of each future step within one centimeter.

1.1.3 Condition-based Maintenance

A Condition-Based Maintenance (CBM) system continuously monitors system parameters in order detect deviations from the expected performance. In some cases, this data is used to make RUL predictions on the system or its components. In an aircraft, this system could monitor the temperature, pressure, environmental, variables of a turbofan and analyze that data to detect when the performance of a component, such as the High-Pressure Turbine, begins to degrade. In this case it would alert the airline and guide the maintenance crew in diagnosing and repairing the engine. This is the approach upon which the HCL model is founded.

1.2 C-MAPPS Dataset

The CMAPPS dataset is a run-to-failure simulation dataset based on a MATLAB and Simulink model of large turbofan engine [5]. The dataset is separated into four main subsets, FD001, FD002, FD003, and FD004. They are separated according to the number of operating conditions and fault-modes present in the data. The six operating conditions ranging from sea-level to 40,000 feet and the fault-modes relate to the degradation of the High-Pressure Compressor (HPC) performance and degradation of the fan's performance. The datasets are further comprised of the sensor data for a number of engines operating through a number of cycles, and the total number of cycles the engine operated for until failure. The dataset also comes pre-split by the total number of engines into training and test sets. These details and divisions of each dataset is summarized in Table 1.

Table 1: Dataset division, fault-mode, and operating condition summary.

Dataset		FD001	FD002	FD003	FD004
Number of Training Engines		100	260	100	248
Number of Test Engines		100	259	100	249
Operating Conditions		1	6	1	6
Fault Mode	**HPC**	YES	YES	YES	YES
	Fan	NO	NO	YES	YES

The data for each engine is a comprised of 26 sensor readings from those that would be available in a typical commercial turbofan engine. However, Amin's model uses only 14 of these sensors that according to previous research, had meaningful impacts predicting RUL [2]. The sensor data used to train the model are listed in Table 2.

Table 2: Data features used to train and test APH model [2].

Sensor Number	Data Type
2	Total temperature at LPC exit
3	Total temperature at HPC exit
4	Total temperature at LPT exit
7	Total pressure at HPC exit
8	Physical fan speed
9	Physical core speed
11	Static pressure at HPC exit
12	Fuel flow ratio to Ps30
13	Corrected fan speed
14	Corrected core speed
15	Bypass ratio
17	Bleed enthalpy
20	HPT coolant bleed
21	LPT coolant speed

The data in Table 6 are know as the features when used as the input the Neural Networks (NN), as discussed in section 1.3.

1.3 Artificial Neural Networks and Machine Learning

1.3.1 Basics of Neural Networks

Artificial Neural Networks (ANN) are essential computer algorithms that have the ability to improve themselves as calculations are performed on each input of data. The core mathematics of a neural network involve matrix-multiplication, linear equations, and simple piecewise or trigonometric functions. The model is built in layers of column vectors or matrices, where each element of the vector or matrix is referred to as neuron. The simplest network is comprised of three layers, an input layer, a hidden layer, and an output.

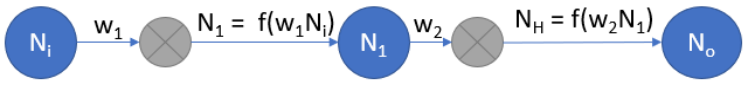

Figure 2: The most basic neural network configuration.

As shown in Figure 1, each layer is connected to following layer by a thread with some random weight. The neural network operates by first passing a know value to the input neuron, N_i. To pass a value to the hidden neuron, N_1, some mathematical operation is performed on the value in N_i with the result being N_1. The same process is repeated to find the value at N_o, the output neuron. This value is the prediction. The prediction is compared to the expected value and the difference is know as the loss. The loss is then used to adjust the values of the weights such that loss would equal zero, meaning the neural network would make perfect predictions. However, the input data is never the same, so the weights that result in a loss of zero for one input will not necessarily result in a loss of zero for another. The goal then is to minimize the loss for the great number of variables.

Sophistication is added to model through the addition of multiple neurons in the hidden layers, functions of which the out becomes the value in the connected neuron. The input layer has as many neurons as needed such that the number of neurons equals the length of the input data. The output is sized according to the data on which the network is making a prediction. There is no limit to the number of hidden neurons or hidden layers, but they are generally limited due to computational concerns. A bias neuron can be added before subsequent layers. When bias neurons are used, the formula between two neurons becomes,

$$N_1 = w_1 * N_i + B_1 \quad (1)$$

If there is a wide range in input values, a function can be used to limit the output of $w_i*N + B$ to a certain range of values. These functions are known as activation functions and their general formula to find N_1 is,

$$N_1 = f_{act.}(wN_i + B) \quad (2)$$

Finally, the connections between one layer and the next is not limited to one. If every neuron in a layer is connected to every neuron in the preceding layer, that (second) layer is referred to as a densely connected layer. This is visualized in Figure 2.

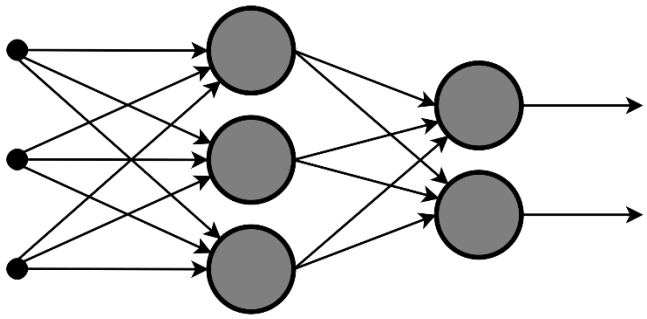

Figure 3: Example of a densely connected hidden layer, and a densely connect output layer [6].

The mathematics connecting the neurons are the same, but in a densely connected neuron, its value is the sum of all the values coming from each neuron feeding into it from the previous layer.

Neural Networks learn by back-propagation of error through the network to change the weights assigned at the beginning of the training session. This involves differentiation along each path between each input neuron and each output neuron. It is beyond the scope of this thesis to explain its operation further. However, the algorithm that determines how this done is called the optimizer. To optimize training results and training time multiple inputs can be given to the model. The loss from each input is then averaged and that average is used to adjust the weights. The number of inputs between weight updates is called the batch size. When all the batches have been put through the model, one training epoch is finished. In complex models, the model may not have learned all that it can. In this case multiple epochs can be used until a predetermined stopping point is reach.

1.3.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a type of network where the outputs of each layer are feature maps built by applying filters to input data. CNN excel at feature recognition and, according to Saha, "…able to successfully capture the spatial and temporal dependencies in an image… [7]" If the input data is a 9 x 9-pixel photo, it can be represented a 9 x 9 matrix with cell values corresponding to color or grayscale value of the photo. The filters are square matrices, typical sized 3x3 or 5x5 filled with some pattern of ones and zeros. The filter is placed over a section of the photo so that all the cells in the filter are within the bounds of the photo. The pixels in the photo are multiplied by the corresponding filter value resulting in either the pixel value or a zero and then all the results are summed and stored in a new matrix. This is

repeated for ever position in the filter can take in the matrix, creating a feature map corresponding to the sum at each filter position.

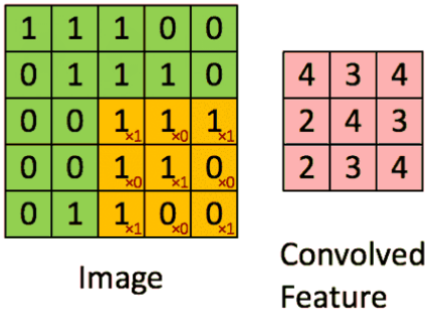

Figure 4: A 3x3 feature map (convolved feature) created by convoluting a 5x5 image by a 3x3 filter [7].

Figure 3 represents a simple convolution of an image into a feature map. The values in the feature map can then be connect to an output layer make a prediction on the data. CNNs are limited to photos, they can used with any data, but they excel at extracting features from input data. The number of filters passed over each data input is limited only by the practicality of that number. Eventually there won't be any benefit to adding more layers, and it may actually end up hurting the model. Furthermore, there is no limit to the amount of convolution steps performed on the data. In this case, filters are applied to the feature maps creating a feature map of the feature map. Normally this would eventually reduce the data to only a few data points, the input data can be padded around its borders to increase the number of positions available to the filter.

1.3.3 Long Short-Term Memory Layers

There is a specific type of machine learning model known as Recurrent Neural Networks (RNN). RNN are similar to the model in Figure 2 except there are usually more than one hidden layer and there are connections that also feed back to previous layers. Instead of the input to a layer comprised of only the sum of values passed to it from previous layers, weighted connections from subsequent layers also produce values that feed into the layer. A Long Short-Term Memory (LSTM) layer is a RNN with the addition of a memory cell that stores information about the data in the LSTM layer previously and the design of the LSTM layer allows it to hold onto that information for some arbitrary amount of time. The information in the memory cell allows the LSTM layer to make temporal connections between subsequent inputs. This makes LSTM layers well suited to time-series data. There is no limit to the number of neurons, but eventually either no further performance can be reach, or the computation cost outweighs the benefits of adding more neurons.

1.3.4 Input Layer Design

The input layer is where the information used to make prediction is loaded into the model. The size of the input layer is determined by the size of the input data and remains fixed throughout the lifetime of the model. The individual data points are known as the features. In

time-series data it useful to design the model to take in multiple timesteps worth of a data. The number of time steps chosen is known as the window size. In the APH model the window size can vary between the model so a variation in size will be tested to find the optimal number. The number of data features is multiplied by the window size to find the total input dimension. In the original APH model, the timestep chosen was limited by the lowest number of engine cycles in the testing data. However, this is accounted for in the design algorithm by dropping engines from the dataset and test sets, that do not meet the window size criteria. The algorithm designed for this thesis also adds a third dimension to the input by changing the number of engines included in the dataset. Finally, all of the model parameters that can be changed by the user are know as hyperparameters. This is in respect to the parameters of the model, the weights connecting neurons, which are not able to be changed by the user directly.

1.3.5 Model Accuracy Scoring

The ability of the model to predict the RUL of an engine is the primary purpose the APH model. To gauge that ability, the predictions made by the model are scored using Root Mean Squared Error (RMSE) of predictions made with the test portion of each dataset.

$$RMSE = \sqrt{\frac{1}{n}(x_1^2 + x_2^2 + \cdots + x_{n-1}^2 + x_n^2)} \tag{3}$$

The RMSE score is effectively the absolute value of the average difference between the predicted RUL and the actual RUL. A score of zero is preferred as it would indicate the model predicts engine RUL perfectly. A score of 1 would indicate that the model's average prediction was correct to within 1 cycle of the engines true RUL. From Figure 4 though, it can be seen that the best model in literature on average made predictions within 12.2 cycles of the true RUL.

2. Algorithm Design

2.1 Problem Statement

In attempting to learn how the model operated internally, as well learn the fundamentals of machine learning and ANN's, it was discovered that it is effectively impossible to replicate any previous results. The reasons for this will be discussed in more detail in the next section, but the underlying source was the genetic optimization algorithm used to tune the models' hyperparameters. Because of these difficulties, the main goal of this thesis shifted from creating a new model, into developing an algorithm for the existing APH model that enabled the user to generate consistent results. With these results, it was theorized that distinct trends can be found within hyperparameter combinations. Which could then be used to guide the evolutionary algorithm of original APH model.

2.2 Objectives and Requirements

The main objective of the design algorithm is that the results generated shall show clearly discernable trends in RMSE, attributable to changes in a specific hyperparameter or a specific hyperparameter combination. This requirement will be considered met, if in isolating a single variable, smooth and continuous changes, or no changes, are observed in the time and/ or accuracy results. It should be noted that it is important to find the variables that have no effect on accuracy and time, as they can be removed from future iterations. Removing variables with no discernable effect allows for a reduction in permutations, thus computation time.

The second objective of the algorithm under design is to generate consistent results in conjunction with the APH model designed by Amin [2]. The need for this arises from the complexity and randomness of the output data, stemming from the optimization algorithm used to optimize hyperparameters of the APH model. It was determined that to improve the consistency of the results generated by the model, the effect each hyperparameter has on the models' performance needed to be more effectively isolated to the adjustments of individual hyperparameters. The main objective of the design algorithm contains two conditions, both of which have to be met. First, the algorithm shall use a consistent and broad set of hyperparameters. Second, the algorithm generates consistent RUL predictions between training runs, for a given dataset. Condition A shall be confirmed if the hyperparameter permutations used are repeatable between training runs, while using multiple options from 8 out of 10 of the hyperparameters listed in Amin's Master's thesis. Condition B is confirmed if the algorithm produces RUL predictions with consistent Root Mean Square Errors (RMSE) between training iterations.

Finally, though not as import for the purposes of this thesis, the model should perform comparable to Amin's [2] genetic algorithm. This is not strictly necessary, as the main objective is to find relationships between hyperparameter selection and accuracy and/or time. It would, however, serve to validate the assumptions made in designing the algorithm.

2.3 Trade-off Study

Two alternate methods to generate meaningful results were considered. The first method was the original method developed by Amin [2]. This involves running the APH model over and over again, tying to generate enough data such that total sample size was statistically significant. The major problem with this approach is that each iteration for the smaller datasets, FD001 and FD003, can take up 2 hours to train on, and the FD002 and FD004 over 4 hours. Furthermore, there is no guarantee that the performance is repeatable. This issue persists on multiple levels. For example, the genetic algorithm was manipulated to retrain one specific model, for 100 separate training iterations.

Table 3: Average, minimum, and maximum RMSE of a single model, retrained 100 different times.

Minimum RMSE	Average RMSE	Maximum RMSE
36.5	43.7	50.0

Table 3 shows that the variance in RMSE is great 6. This means that on average a single model may guess within 6 cycles of the actual RMSE. This is due to the way the model weights are randomly generated when it is initialized.

Table 4: Minimum accuracy out of 263 different models.

Minimum Accuracy	Totals Models Trained
48.8	263

Table 5: Minimum accuracy out of 479 different models.

Minimum Accuracy	Total Models Trained
17.6	479

The data in Tables 3 and 4 were both generated during training on dataset FD001. The large variance in RMSE scores further demonstrates the difficulties in trying to derive associations between hyperparameters and training scores.

The second approach considered was a brute-force training regime. This approach seems ideal as it attempts every permutation of hyperparameters. Amin [2] selected 10 different hyperparameters, with a range between 3 to 6 options for each hyperparameter to be evolved for in the algorithm [2]. In addition, the model had the capability to change the window size of the input tensor. This adds several more options for permutations. A rough estimate puts the total permutations at nearly 4,000,000. At an average time of 10 seconds per model, the total time to calculate all permutation would be about 1.25 years! This is clearly not a feasible approach given equipment availability and time constraints.

Given the numerous possible permutations of hyperparameters, it is easy to see why any given training run, using Amin's [2] genetic algorithm and AHP model, can finish with high RMSE scores. One training run of 30 generations, each comprised 30 unique genomes would only test 900 different permutations, a tiny fraction of the possibilities. The genetic algorithm trades off certainty in finding the best solution for the ability to find a good solution in a fraction of the time. This is desirable as it shows capability for deployment in a wide range of PHM applications with out the need for a years and three months' worth of computation time. Furthermore, there is no guarantee that the best solution will be significantly better than those generated by the genetic algorithm meaning the time investment may not return a significant accuracy gain. Finally, both approaches may be fundamentally limited by the design of the model and abundance of data. This strengthens the need for a more controllable algorithm, that has the ability to find trends in the data as it can help determine the maximum model performance. Ideally the design algorithm would reduce the total number or variables to an

amount where a computer could go through every remaining permutation in a reasonable amount of time.

2.4 Design Process

2.4.1 Preliminary Hyperparameter Selection

Although there was a clear need for a more ordered approach to model performance tuning, some meaningful insights were discovered during the original study of the APH model. One of the hyperparameters used in the genetic algorithm was the number epochs allotted for model to train. In the genetic algorithm a range of total epochs was chosen from 200-600, in 100 epoch increments.

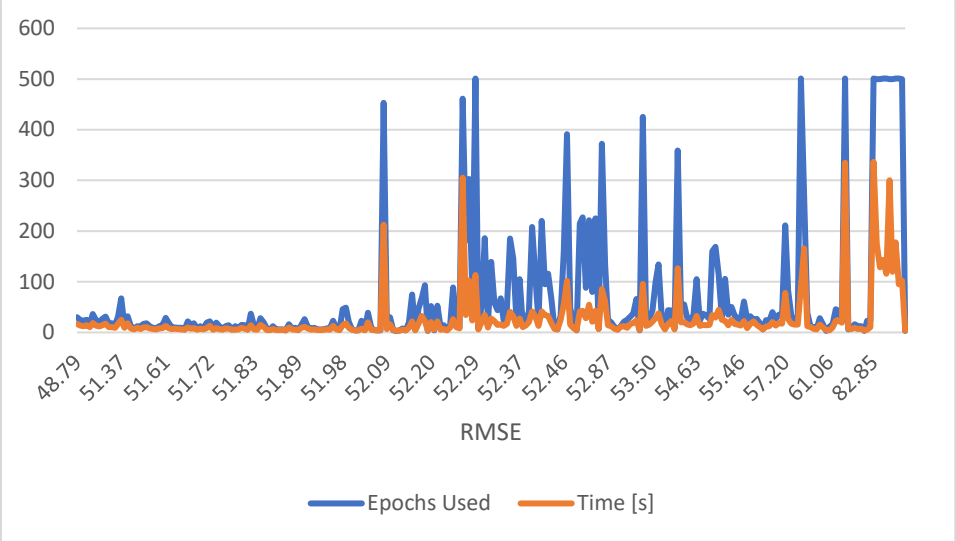

Figure 5: RMSE scores relative to training time and epochs used, example 1.

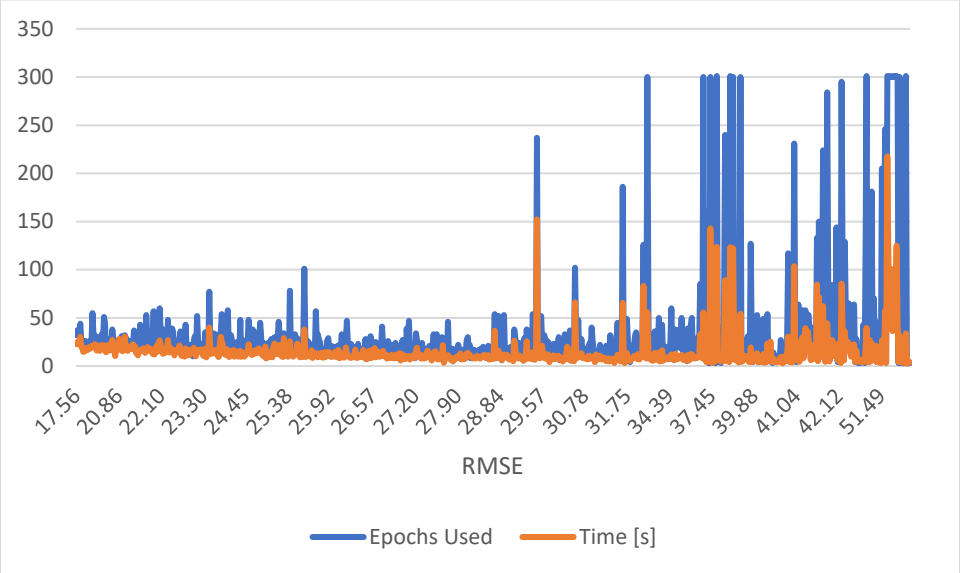

Figure 6: RMSE score relative to training time and epochs used, example 2.

Figures 2 and 3 demonstrate that regardless of the overall performance of the training run, the best performing models never required the entire allotted epochs. In fact, models that used all of the allotted epochs were almost always among the worst performing, as gauged by both RMSE score and length of training period. Figures 2 and 3 show that the epoch variable can be removed from rotation and set to a constant value. The first design choice was to limit epochs to 50 as no best performing model required more than 50 epochs to train. This ability is directly attributable to the Early Stopping callback built into the AHP model. The early stopping callback is designed to stop training when the validation loss metric changes by less than 10% between epochs. This limits overfitting the data while also limiting maximum training time. By extension it renders the epoch variable obsolete except in cases where model progress enough to avoid early stopping but slow enough that their computation far outweighs any potential benefit.

Further study of the data generated using Amin's [2] genetic algorithm and AHP model show that the dropout value has little effect on accuracy.

Table 6: Average RMSE relative to various dropout values; FD001

Dropout	Average RMSE
0.3	19.15163254
0.2	19.39653863
0.4	19.42844662
0	19.72313942

Table 7: Average RMSE relative to various dropout values; FD003

Dropout	Average RMSE
0.3	22.07
0.4	23.22
0.2	23.29
0	24.14

Tables 5 and 6 are just a sample of the data used to make this determination, but the results are similar across every training run. Although the effect is almost negligible, the dropout value of 0.3 consistently has the lowest average RMSE scores. For this reason, it was decided to set the dropout value to a constant value of 0.3 in further testing.

2.4.2 Final Hyperparameter Selection

After setting these two values as constants, little else was gleaned from further study of the data, as generated using Amin's [2] genetic algorithm. Removing the allotted epoch number and dropout parameters from consideration decreased the breadth of hyperparameter selection, so to satisfy Condition A of the first objective, two more variables were selected for analysis. The added hyperparameters selected were the number of engines and window-size. These bring the

total number of individual hyperparameters to 10; the values and types of which are listed in Table 6.

Table 8: Hyperparameter selection table, grouped by purpose, listed by order of testing.

	Hyperparameter	Value/ Type
Feature Selection	CNN Activation Function	['relu', 'tanh', 'sigmoid']
	Number of CNN Filters	[10, 16, 32, 64]
	Number of CNN Layers	[1, 2, 3, 4]
Time Dependency	LSTM Activation Function	['relu', 'tanh', 'sigmoid']
	Number of LSTM Neurons	[10, 30, 50, 80]
	Number of LSTM Layers	[1, 2]
Model Optimization	Optimizer	['rmsprop', 'adam', 'adagrad', 'acadelta']
	Batch-size	[200, 400, 500, 600]
	Learning-rate	[0.1, 0.01, 0,001]
	Number of Engines	[1, 2, 3]
	Window Size	[10, 20, 30, 40]

The APH model consists of two distinct parts. The CNN layers, whose purpose is to extract features, and the LSTM layers, whose purpose is to extract time-dependencies between data inputs. Table 7 groups the hyperparameters according to those two parts, with the third group distinguished by there effect on the model as a whole. Within those groups, the hyperparameters are listed in the order by which they will be varied as the algorithm progresses.

The grouping and order of the hyperparameters were chosen according to the requirements of the second objective. The grouping isolates changes in the effectiveness of the CNN layers to extract features, in the effectiveness of the LSTM layers to extract time-dependencies, and in the effectiveness over overall model optimization hyperparameters. The APH model was built with sequential layers, so the ordering was chosen to reflect the flow of data through the model. The general idea was to first build a CNN model that best extracts data-features, then build an LSTM model that best determines time-variations between the data-features. Following the construction of individual models, the performance as whole was optimized by testing the hyperparameters that were thought to have the least effect on individual model variations. In keeping with the data flow hierarchy, the ordering of the Model Optimization group began with testing individual optimizers. Finally, last four hyperparameters, batch-size were then tested individually with the intent of fine-tuning the model performance, in an effort to satisfy the final objective.

2.4.3 Design Assumptions

Several assumptions were made in the determining the hyperparameter optimization groupings and orderings. The first assumption was that the performance of the CNN portion of the model was correlated with the performance of the LSTM. This can be verified by comparing the performance of the model during the LSTM optimization stage to the performance of the

corresponding subset of model configurations during the CNN optimization stage. If the performance changes between individual configurations are linearly correlated, then the performance of the LSTM model is not affected by the performance of the CNN model. If the results are random, then the CNN model does not reliably enhance the effectiveness of the LSTM model. Finally, if the performance of the LSTM model is always better, and in an exponential manner, the effectiveness of the LSTM model is indeed enhanced by the CNN model. The second assumption is that the performance of any individual configuration of the model is not correlated to the specific use of any individual hyperparameter in the Model Optimization group. This assumption can be verified by comparing the performance changes between configurations generated before the Model Optimization stage, to the performance changes during testing of each Model Optimization hyperparameter. If the changes are consistent between model configurations, then the assumption that the Model Optimization are only general optimization and not specific to individual model configurations holds true. For example, if each iteration of the optimizer stage are ranked by performance, the rankings shouldn't change after the batch-size optimization stage.

2.4.4 Algorithm Operation

The overall design of the algorithm is quite simple, but this is necessary to achieve the desired outcome. Each odd numbered stage creates of a list of all the permutations of the hyperparameters associated with that stage. The top ten performing permutations are then passed to an even numbered stage. In the even numbered stage, the top 20 genomes are retrained 5 times each, with the top 10 performers passed to the on to the following stage odd numbered stage. The second step decreases the risk that a good model is passed over because it had a bad training run, while decreasing the number of models that need to be train in an odd number stage.

In this manner, each possible configuration, of up to four CNN layers was tested, and every possible configuration of up to 2 LSTM layers. Following this each separate optimizer was tested, with the top 20 performing configurations passed to the retraining stage, and the top 10 moving to the batch-size optimization stage. The model continues until the window-size stage, at which point the results are output to the Google Colab Notebook and saved to an excel file.

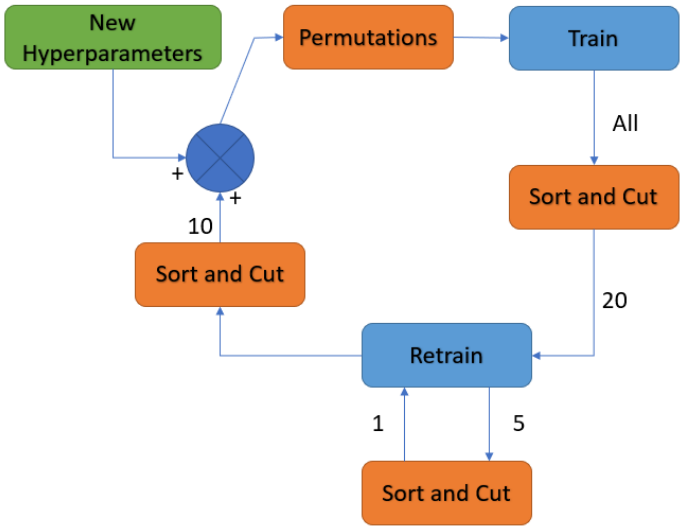

Figure 7: Training and data generation algorithm

Figure 4 is visualisation representation of the training algorithm and data throughput at each stage. The numbers by certain paths represent the total unique permutations that are passed to the next stage.

3. Results and Analysis

The results generated in this section were generated using only the FD001 dataset. Difficulties arose during the design process that limited the ability for the author to produce more results. Although this may reduce confidence in the ability of the design algorithm to extrapolate meaningful data outside the FD001 dataset, the design was not, in any way, tailored specifically for the FD001 dataset. As a result, the author is certain the results contained in this section are representative of future results generated with datasets FD002,3 and 4, or with any other machine learning model.

3.1 CNN Layers - Filters and Activation Functions

Section 3.1 explores the possible connections between RMSE score, the number of filters in a CNN layer, and the activation function. In each example, one hyperparameter was allowed to vary while the others were held constant.

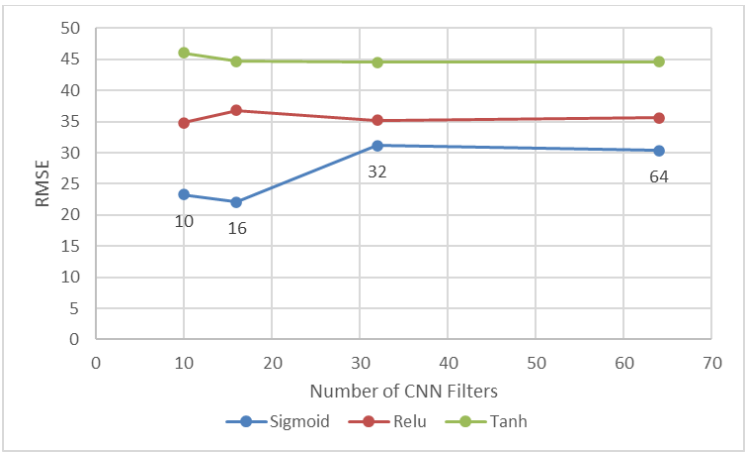

Figure 8: Effect of changing filter size and constant CNN activation function.

Figure 8 shows a clear link between the RMSE score and activation function. The sigmoid function enables the CNN layer to achieve the best performance. However, there does not appear to be a link between the number of filters and performance of the CNN layers. This doesn't mean there is not any correlation, rather it suggest that there are too few datapoints for a distinction to be made.

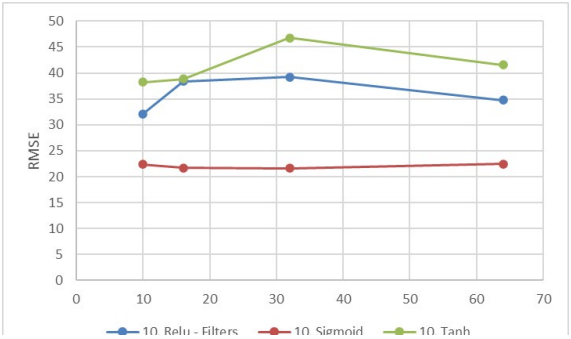

Figure 9: Effect of CNN activation functions and 10 filters, with a single CNN layer.

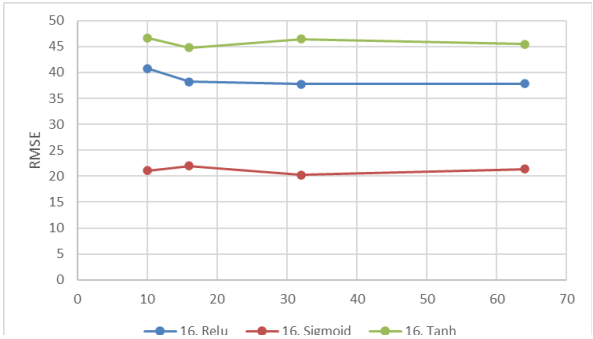

Figure 10: Effect of CNN activation functions and 16 filters, with a single CNN layer.

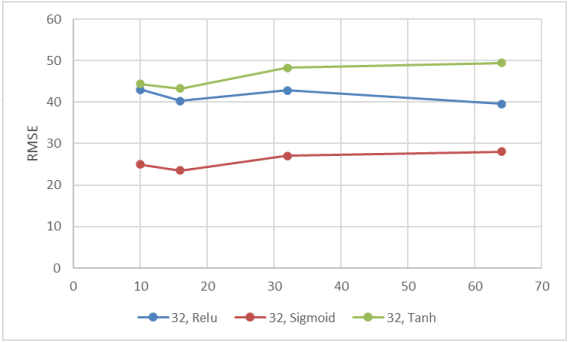

Figure 11: Effect of CNN activation functions and 32 filters, with a single CNN layer.

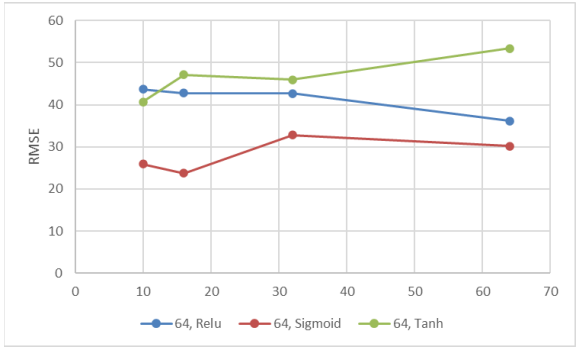

Figure 12: Effect of CNN activation functions and 64 filters, with a single CNN layer.

The HCL models used in Figures 8-11 were compiled with the following constant hyperparameters:

- A single CNN layer
- A single LSTM layer with 80 neurons and using *tanh* activation function
- Window-sizes of 30 cycles
- Batch sizes of 200
- The *rmsprop* optimizer
- A learning rate of 0.1

The only variation between each of Figures 8 to 11 is the number of filters used to build the feature maps. In each of Figures 8 to 11, the number of filters is held constant, and the colour of the line corresponds to the activation function used with the CNN layer. It is clear from Figures 8 to 11 that the sigmoid function helps the achieve the best RMSE scores regardless of number of filters used. The *relu* and *tanh* functions rank 2nd and 3rd in nearly every scenario.

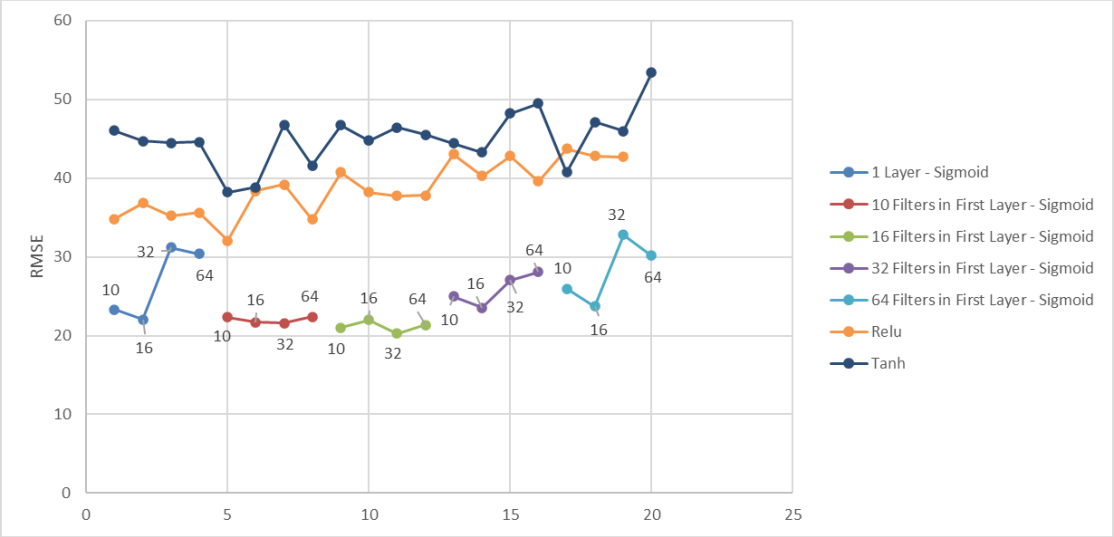

Figure 13: Effect of CNN activation functions with varying numbers of filters.

Figure 12 graphs the RMSE scores of each permutation of the available CNN hyperparameters. The groupings in the Sigmoid layer (bottom row) correspond to groups whose members all had 10 filters used in their first (or only) layer. Travelling in the positive vertical direction, from any point in the sigmoid layer, one will arrive a at point on either the *relu* curve or *tanh* curve where the only variation between hyperparameter selection. Figure 12 provides further evidence that the using the sigmoid activation allows the CNN layer to achieve their maximum effectiveness. The *sigmoid* layer of Figure 12 also shows a weak in connection between the number of filters used in each layer and the model's RMSE score, with presence of a global minimum in the model using two layers, with 16 filters in the first layer and 32 filters in the second layer.

However, with only one data point for each permutation, and given the results in Tables 4 and 5 (large possible variation in individual model performance), the connection is tenuous at best.

3.2 LSTM Layers – Neurons and Activation Functions

Section 3.2 expands upon the testing performed in section 3.1. In this section, the links between the RMSE score, number of neurons in a LSTM layer, and the activation function of choice was explored. Furthermore, the number of layers was varied to gauge whether any trends continue as the number of layers was increased. In each example one hyperparameter was changed while the rest were held constant. The top 5 configurations from the CNN tuning stage were carried forward to the LSTM tuning stage and train on each of the 60 unique LSTM layer configurations.

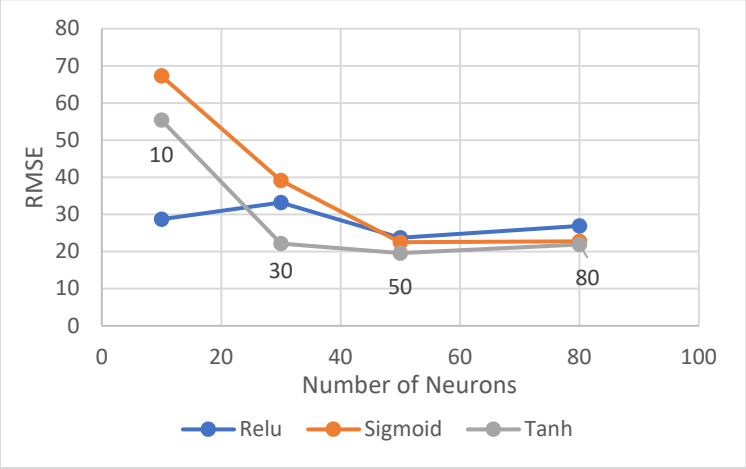

Figure 14: Effect of activation function and number of neurons on RMSE score.

Figure 14 displays the effect of increasing the number of neurons, relative to each of the LSTM activation functions, had on RMSE score, when a single LSTM layer was selected. The plots in Figure 14 suggest that the choice of activation function had little effect on the performance of the model. There was, however, a much stronger correlation between the number of neurons and the performance of the model. The model with 50 neurons using the *tanh* activation function performed the best overall, but the results of Table 4 and 5 suggest that there may be enough variation in RMSE scores that this conclusion may prove incorrect if each model is all given more chances to train.

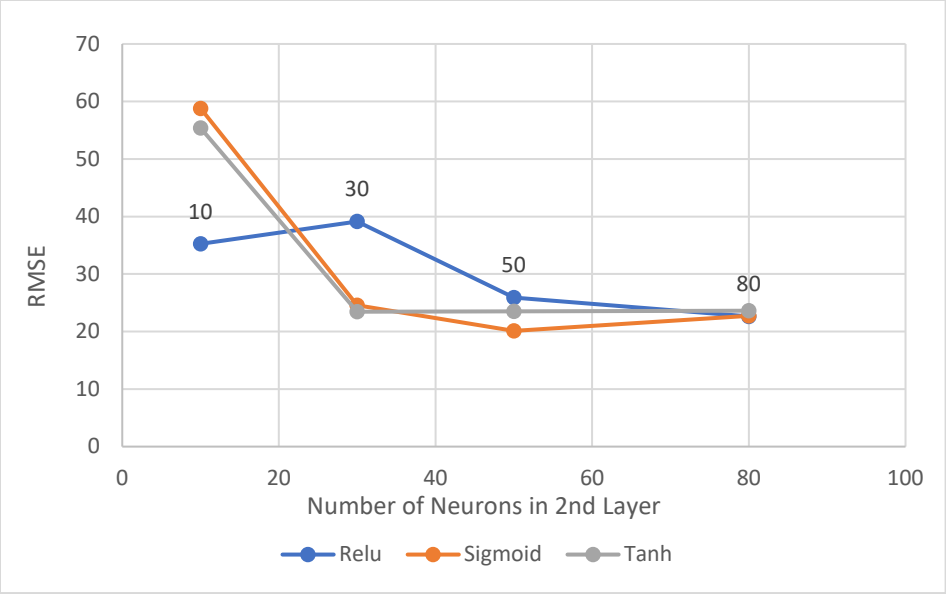

Figure 15: Effect of increasing number of neurons in 2nd LSTM layer, with 10 Filters in CNN layer and 10 Neurons in 1st LSTM layer.

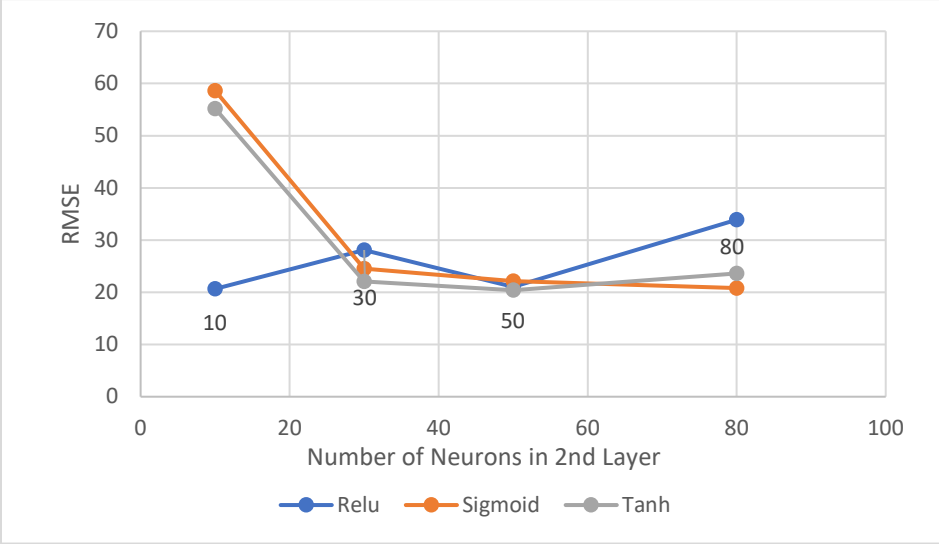

Figure 16: Effect of increasing number of neurons in 2nd LSTM layer, with 10 Filters in CNN layer and 30 Neurons in 1st LSTM layer.

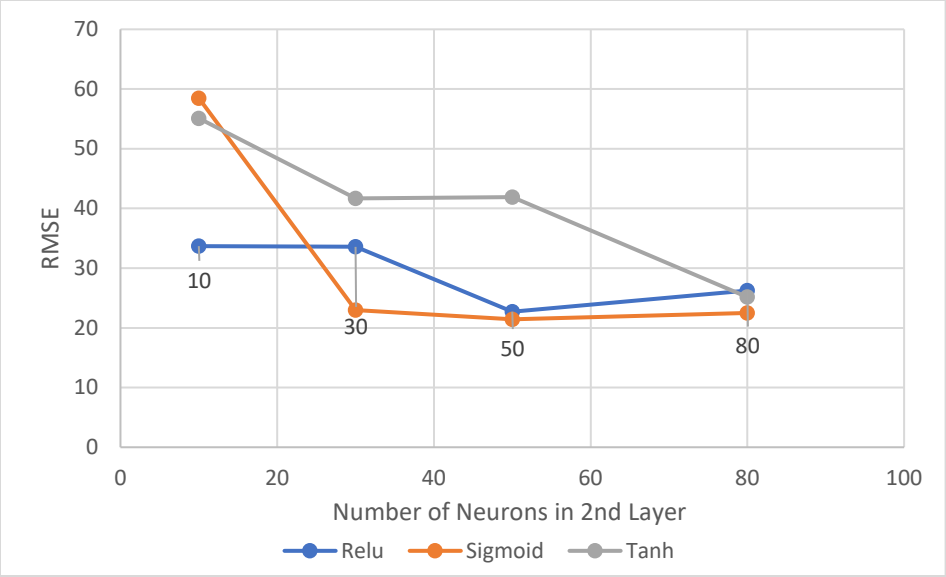

Figure 17: Effect of increasing number of neurons in 2nd LSTM layer, with 10 Filters in CNN layer and 50 Neurons in 1st LSTM layer.

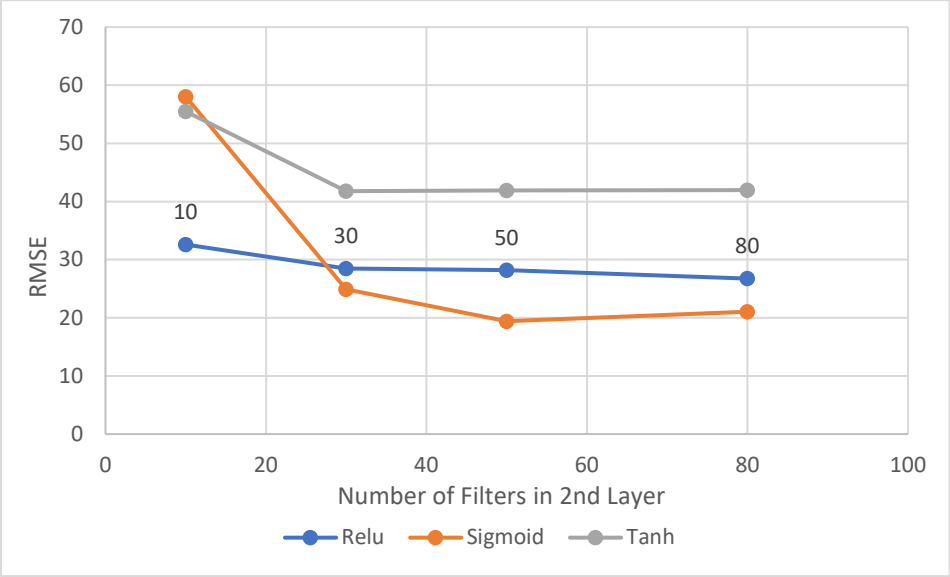

Figure 18: Effect of increasing number of neurons in 2nd LSTM layer, with 10 Filters in CNN layer and 50 Neurons in 1st LSTM layer.

Figures 15-18 compare the effects of the three activation functions when there is a second LSTM layer, with an increasing number of neurons. A significant trend exist between a low number of neurons in an LSTM layer and the performance of the *relu* function relative to the *sigmoid* and *tanh* functions. This suggests in cases where computation time (less filters means less calculations) is prioritized over accuracy, the *relu* function will achieve the best results. In the 2 CNN layer, 2 LSTM layer configuration this effect became most pronounced in Figure 16, where

30 filters in the CNN layer were used with an LSTM layer of 10 neurons. Comparing Figure 16 to Figure 14 shows that there was a decrease in RMSE score of 10, just by adding a second 10-neuron LSTM layer. However, overall, there was not a significant performance gain by adding a second LSTM layer.

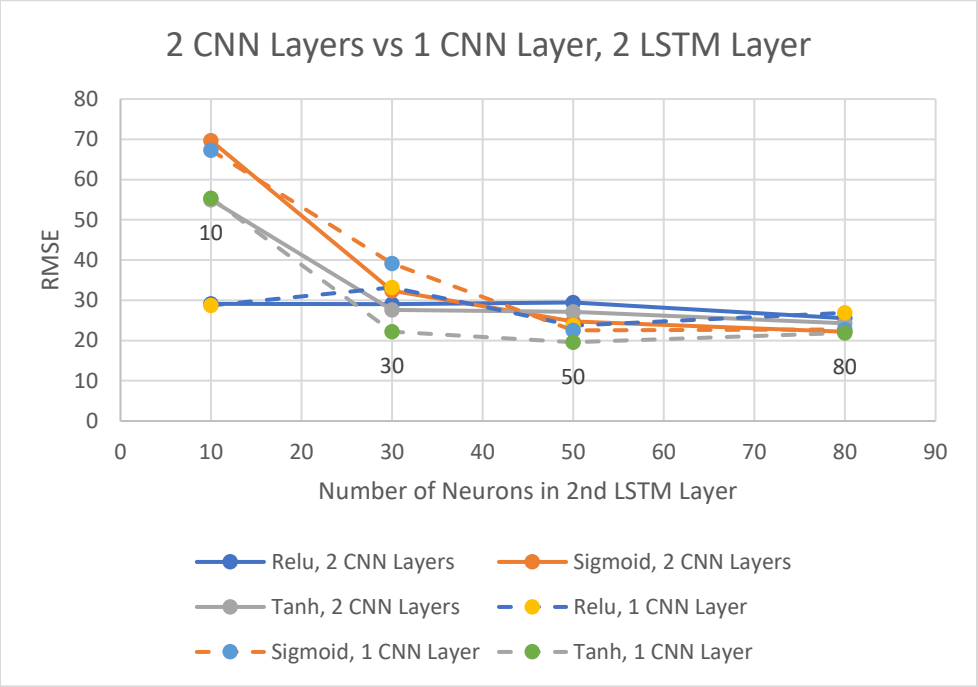

Figure 19: Effects of adding a second CNN layer and a second LSTM Layer.

The results shown in Figure 19 compare a single CNN layer, single LSTM layer HCL model with a double CNN layer, double LSTM layer HCL model. The results in Figure 19 were generated using single layer HCL model that used a 10 filter CNN layer, and double CNN layer HCL model using 16 filters in the first layer and 10 filters in the second. In both models the final LSTM layer was increased from 10 and 80 neurons, and in the two-layer HCL model the first layer had 10 neurons. Ultimately, the Figure 19 demonstrates that there was no performance gain inherent in doubling the amounts of unique layers, and that the only consistent gains come from adjusting the CNN filters and LSTM neurons.

3.3 Optimizer

Following the construction and training of each HCL model with unique CNN and LSTM hyperparameters, the top 5 performing models were selected to pass through to the optimizer testing stage. In this stage each model was trained using each of the four unique optimizers.

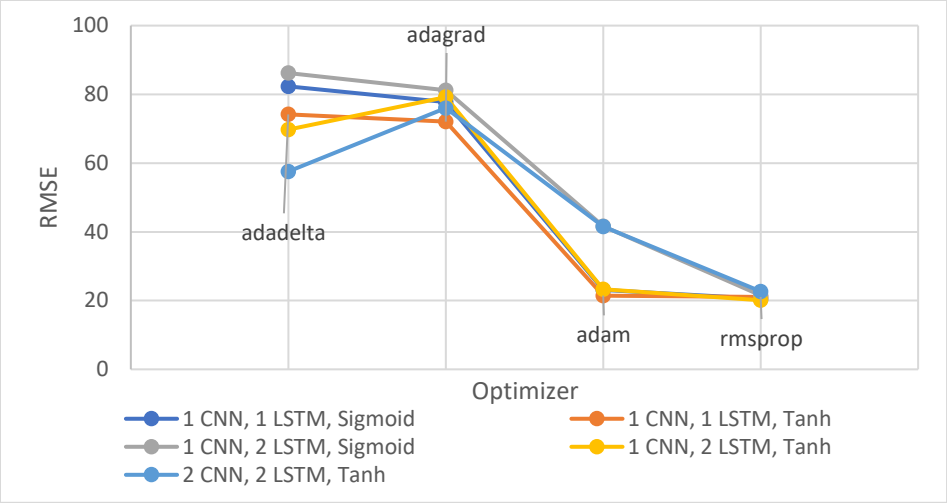

Figure 20: Effects of optimizer on model performance.

Figure 20 compares the results generated by each of the top 5 HCL models with each of the four optimizers. Each of the models used the *sigmoid* activation function, with 10 filters. The lone HCL model with 2 CNN layers used 16 filters in its second layer. Both single LSTM layer models had 80 neurons in their LSTM layer. The single-CNN, double-LSTM layer model using the *sigmoid* activation function had 80 neurons in its first layer and 50 neurons in its second. The other single-CNN, double-LSTM layer HCL model had 10 and 80 neurons in its first and second LSTM layers, respectively. Among the four optimizers, the *adagrad* and *rmsprop* optimizers performed the most consistently, with the *rmsprop* optimizer generating the top three, 5th and 6th best RMSE scores. Unexpectedly, the *adadelta* optimizer performed better, relative to itself, when the *tanh* function is used in the LSTM layers. This suggests that performs better when the output of a neuron maintains its sign, as the sigmoid only produces positive numbers as an output.

3.4 Batch Size

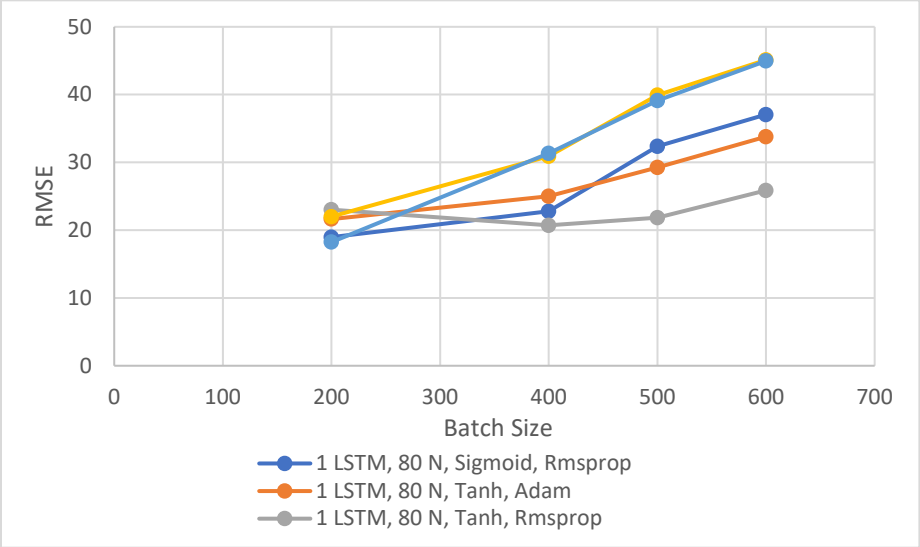

Figure 21: Effect of batch size on performance.

Figure 19 compares the performance of each HCL model as its batch size was increased from 200 to 600. Each model contains a single CNN layer using 10 filters and the sigmoid activation function. The legend contains information pertaining to the LSTM layer(s) parameters, indicating the number of layers, the layer(s)'s respective number of neurons, the activation function and finally the model's optimizer. Except for the single-layer LSTM HCL model with using the *tanh* activation function and *rmsprop* optimizer, each HCL model showed an increase in performance as the batch size was increased. The model that did not, had a local minimum RMSE score with a batch size of 400, suggesting a batch size of 400 was the optimum size for that particular configuration. Given the consistency of the other models' performance with batch size, it would appear that further performance gains could be achieved by testing a lower range of batch sizes.

3.5 Number of Engines

The results of this section were generated by increase the number of engines from which data could be included in each training input. The original HCL model only trained on data input containing information from one engine at a time. The idea was to allow the HCL model to establish patterns more easily in data sourced from different engines enabling greater generalization.

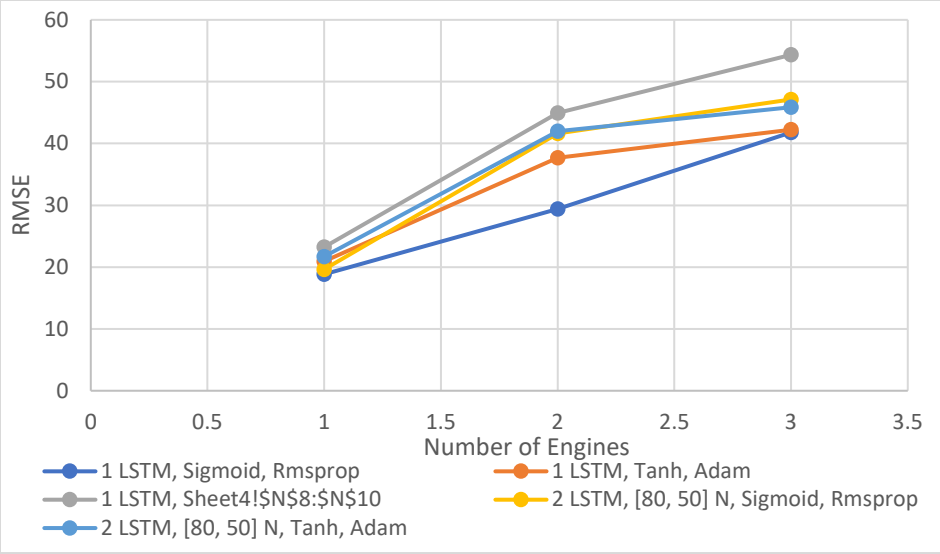

Figure 22: Effect of number of engines in input on performance.

Figure 21 shows that increasing the number of engines in the input data clearly inhibits model performance. This suggests the model may have become over specialized to the input data or that the idea is fundamentally flawed.

3.6 Window Size

This section varied the size of the input data tensor. For time-series data, the window size is the number of time dependent cycles of data given to input into the model during training. For this stage of optimization, the window size ranged from 10 to 40 cycles in 10 cycle increments.

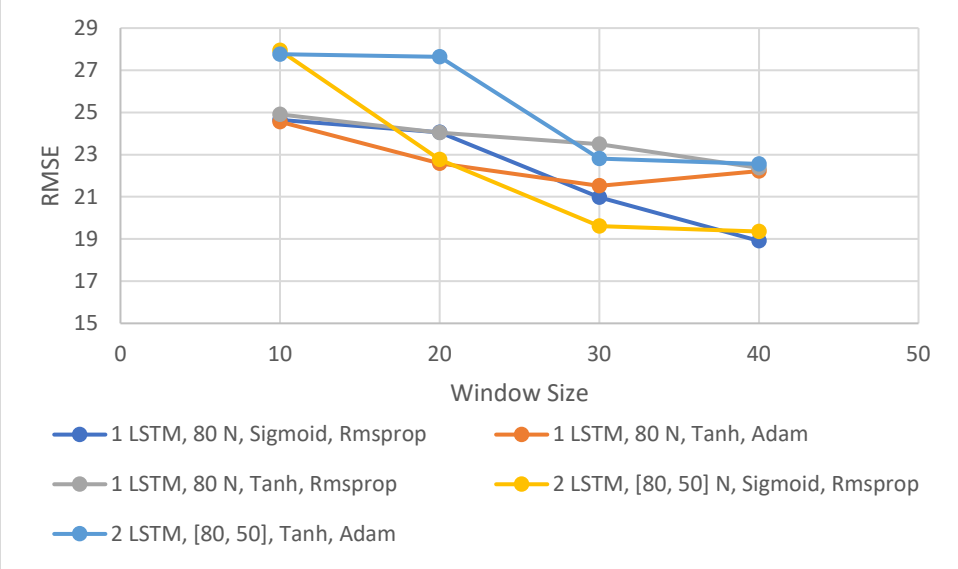

Figure 23: Effects of window size on performance.

Figure 22 shows a that there was a correlation between window size and performance. In all cases, increasing the window size improved performance, however, the shapes of each graph indicate whether further increase will improve performance. Both double-layer models indicate that they are at or very near local minima, thus further increases will not improve performance. The curve of the single-layer LSTM model indicates that it the rate of increasing performance is decreasing as the window size grows. This suggests that further increases in window size would allow that HCL model to achieve better results. The curve of the single-layer LSTM model utilizing the *rmsprop* optimizer and *tanh* activation function appears to have an increasing slope suggesting that further increases in window size will result in greater relative gain. Finally, in the curve of the single-layer LSTM model utilizing the *adam* optimizer, there is a local minimum indicating further increases in window size will result in decreased performance.

4 Conclusion

There were three objectives established in section 3.1, of which the second objective had two qualifying conditions. The main objective and second objective were design requirements, while the third object was a design goal.

1. The results generated using the design algorithm shall generate results that show clearly discernable trends in RMSE, attributable to changes in a specific hyperparameter or a specific hyperparameter combination.

2. The algorithm under design shall generate consistent results in conjunction with the APH model designed by Amin [2].
    a. The algorithm shall use a consistent and broad set of hyperparameters.
    b. The algorithm shall generate consistent RUL predictions between training runs, across all four datasets.

3. The algorithm under design should generate results comparable to those generated by Amin outlined in Figure 1.

There is overwhelming support that the requirements of Objective 1 were met. The design of the algorithm only ever alters one hyperparameter while keeping the remaining hyperparameters constant. This is reflected in Figures 14-22 all showing discernable trends in the results allowing for simple validation of the effect of a hyperparameter or lack thereof. There were limited cases of complex or seemingly random variations in performance, such as the effect of the *adam* optimizer in Figure 20, but even that result showed discernable trends correlating to the changes in hyperparameters permutations. Therefore objective 1 was met.

The confirmation of Condition A of Objective 2 is easily verified by the design of the algorithm itself, which uses lists of predetermined hyperparameters. The ordered manner in which the lists of hyperparameter permutations are generated, using “for loops” further satisfies Condition A. The requirement for a broad array of hyperparameters, in Condition A is verified

by evidence of the use of the 8 of 10 of the hyperparameters used in Amins genetic algorithm [2], with the addition of two more, by the author. Condition B was designed for, by including a retraining loop between each hyperparameter testing stage. However, a lack of repeated results precludes the verification of Condition A. Therefore, objective two can only be considered partially met.

Finally, Objective 3 was not met. The author was unable to replicate the performance obtained by Amin as presented in Figure 1. This was not achieved using either Amin's genetic algorithm or the author's algorithm. However, results below an RMSE of 18 were generated in trial runs of the algorithm.

In conclusion, this algorithm shows promise as a tool to help improve the results generated not only using the genetic algorithm designed by Amin [2], but for any machine learning model with a large number of hyperparameters. The algorithm has shown that individual components of a model can be isolated, tested and tuned eliminating the need to try every possible permutation of hyperparameters. This helps the genetic algorithm immensely as certain hyperparameters, such as the *sigmoid* activation function used in the CNN layers, can be set to a constant. This allows the user to increase to options for specific hyperparameter classes without decreasing confidence in the final result. The algorithm is not perfect though, so a number of suggestions are made for future work, in the next section.

5 Future Work

As mentioned in the preface of section 3, the results in this thesis were generated only using the FD001 dataset. Testing was done using the other three datasets, so there is no reason why the algorithm will not generate similar results with the other datasets. Furthermore, Condition B was not fully verified. More testing should be done on dataset FD001 as well to confirm whether or not the steps taken to reduce inherent model training inconsistency in the algorithm are adequate. More can be done to increase ease of use. The current version of code is effectively a script. The algorithm could be further reduced to function calls and a "for loop" to progress the algorithm through each stage.

References

[1] National Air and Space Administration: Prognostics Center of Excellence, "PCoE Datasets," 2008. [Online]. Available: https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/.

[2] U. Amin, "Prediction of Remaining Useful Life for Aircraft Engines Using Convolutional and Reccurent Neural Networks," Ryerson University, Toronto, 2021.

[3] D. L. Goodman, J. Hofmeister and F. Szidarovszky, "Approaches For Prognosis and Health Management/ Monitoring (PHM)," 2019. [Online]. Available: DOI: 10.1002/9781119356677.ch2.

[4] G. J. Vachtsevanos and K. P. Valavanis, "A Novel Approach to Integrated Vehicle Management," 2018. [Online]. Available: https://www.sto.nato.int/publications/STO%20Meeting%20Proceedings/STO-MP-AVT-305/MP-AVT-305-11.pdf.

[5] National Air and Space Admiistration, "NASA Software: Commercial Modular Aero-Propulsion System Simulation," [Online]. Available: https://software.nasa.gov/software/LEW-18315-1.

[6] Offnfopt, *Multi-Layer Neural Network-Vector-Blank.svg,* Wikimedia Commons, the free media repository, 2015.

[7] S. Sumit, "A Comprehensice Guide to Convolutional Neural Networks," Towards Data Science, 15 December 2018. [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[8] M. ElDali and K. D. Kumar, "Fault Diagnosis and Prognosis of Aerospace Systems Using Growing Reccurent Neural Networks and LSTM," Ryerson University, Toronto, 2020.