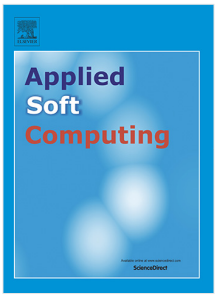
Accepted Manuscript



Optimizing long short-term memory recurrent neural networks using ant colony optimization to predict turbine engine vibration

AbdElRahman ElSaid, Fatima El Jamiy, James Higgins,
Brandon Wild, Travis Desell

PII: S1568-4946(18)30530-1
DOI: https://doi.org/10.1016/j.asoc.2018.09.013
Reference: ASOC 5092

To appear in: *Applied Soft Computing Journal*

Received date : 7 July 2017
Revised date : 1 July 2018
Accepted date : 8 September 2018

Please cite this article as: A. ElSaid, et al., Optimizing long short-term memory recurrent neural networks using ant colony optimization to predict turbine engine vibration, *Applied Soft Computing Journal* (2018), https://doi.org/10.1016/j.asoc.2018.09.013

Optimizing Long Short-Term Memory Recurrent Neural Networks Using Ant Colony Optimization to Predict Turbine Engine Vibration

AbdElRahman ElSaid[a,*], Fatima El Jamiy[b], James Higgins[c], Brandon Wild[a], Travis Desell[a]

[a]*B. Thomas Golisano College of Computing & Information Sciences, Rochester Institute of Technology, Rochester, New York 14623*
[b]*Department of Computer Science, University of North Dakota, Grand Forks, North Dakota 58202*
[c]*Department of Aviation, University of North Dakota, Grand Forks, North Dakota 58202*

Abstract

This article expands on research that has been done to develop a recurrent neural network (RNN) capable of predicting aircraft engine vibrations using long short-term memory (LSTM) neurons. LSTM RNNs can provide a more generalizable and robust method for prediction over analytical calculations of engine vibration, as analytical calculations must be solved iteratively based on specific empirical engine parameters, making this approach ungeneralizable across multiple engines. In initial work, multiple LSTM RNN architectures were proposed, evaluated and compared. This research improves the performance of the most effective LSTM network design proposed in the previous work by using a promising neuroevolution method based on ant colony optimization (ACO) to develop and enhance the LSTM cell structure of the network. A parallelized version of the ACO neuroevolution algorithm has been developed and the evolved LSTM RNNs were compared to the previously used fixed topology. The evolved networks were trained on a large database of flight data records obtained from an airline containing flights that suffered from excessive vibration. Results were obtained using MPI (Message Passing Interface) on a high performance computing (HPC) cluster, evolving 1000 different LSTM cell structures using 208 cores over 21 days. The new evolved LSTM cells showed an improvement of 1.34%, reducing the mean prediction error from 5.61% to 4.27% when predicting excessive engine vibrations 10 seconds in the future while at the same time dramatically reducing the number of weights from 21,170 to 13,150. The optimized LSTM also performed significantly better than traditional Nonlinear Output Error (NOE), Nonlinear AutoRegression with eXogenous (NARX) inputs, and Nonlinear Box-Jenkins (NBJ) models, which only reached error rates of 15.73%, 12.06% and 15.05%, respectively. Furthermore, the LSTM regularization method was used to validate the ACO. The ACO LSTM out performed the regularized LSTM by 3.35%. The NOE, NARX, and NBJ models were also regularized for cross validation, and the mean prediction errors were 8.70%, 9.40% and 9.43% respectively, which gives credit for the ant colony optimized LSTM RNN.

Keywords: Ant Colony Optimization, ACO, Long Short Term Memory Recurrent Neural Network, LSTM, Recurrent Neural Network, RNN, Time Series Prediction, Aviation, Aerospace Engineering, Turbomachinery, Turbine engine vibration, Flight Parameters Prediction
2010 MSC: 00-01, 99-00

1. Introduction

Aircraft engine vibration is of critical interest to the aviation industry, and accurate predictions of excessive engine vibrations have the potential to save time, effort, money as well as human lives in the aviation industry. An aircraft engine, as turbo-machinery, should normally vibrate as it has many dynamic parts. However, it is not supposed to exceed resonance limits as to not destroy the engines [1]. As an example, A. V.

*Corresponding author
Email addresses: aae8800@g.rit.edu (AbdElRahman ElSaid), fatima.eljamiy@und.edu (Fatima El Jamiy), jhiggins@aero.und.edu (James Higgins), bwild@aero.und.edu (Brandon Wild), tdesell@cs.und.edu (Travis Desell)
URL: http://people.cs.und.edu/~tdesell (Travis Desell)

Srinivasan [1] describes vibrations generated from engine blades' fluttering. Engine blades are the rotating engine components that have the largest dimensions among the other engine components. When rotating at high speeds, they will withstand high centrifugal forces that would logically give the highest contribution to engine vibrations.

Engine vibrations are not that simple to calculate or predict analytically because of the fact that various parameters contribute to their occurrence. This fact is always a problem for aviation performance monitors, especially as engines vary in design, size, operation conditions, service life span, the aircraft they are mounted on, and many other parameters. Most of these parameters contributions can be translated in some key parameters measured and recorded on the flight data recorder. Nonetheless, vibrations are likely to be a result of a mixture of these contributions, making it very hard to predict the real cause behind the excess in vibrations.

As such, engine vibration is a complex problem depending on an unknown number of parameters interacting over an unknown extended period of time, which poses a challenge in analyzing its causes and triggers. Further, in order to develop sensors capable of effectively warning pilots of impending vibration events, predictions need to be made over longer time scales, *i.e.* for a pilot to be able to react and prevent the event a warning 10 to 20 seconds before the event occurs would be necessary. This presents a challenge as most time series data prediction algorithms focus on only predicting the next reading, which in the case of most flight data, which operates at 1 Hz, would be the next second.

Holistic computation methods represent a promising solution for this problem by letting the computers find relations and anomalies that might lead to the problem through a learning process using time series data from flight data recorders (FDR). Traditional neural networks, however, lack the required capabilities to capture those relations and anomalies as they work on current time series without taking the effect of the previous time instances' parameters on the current or future time instants. Due to this, recurrent neural networks have been developed which utilize memory neurons that retain information from previous passes for use with the current experienced data, giving a chance for the neural network to know which parameter really have higher contributions to the investigated problem.

However, these complicated neural network designs in turn posses their own challenges. Regardless of the difficulty of implementing it to a specific problem, the learning process is the main concern when dealing with such neural networks with a large number of interactive connections. When supervised learning is considered and the back-propagation is implemented to update the weights of the connections of the neural network, vanishing and exploding gradients are very serious obstacles for the successfully training recurrent neural networks. As noted by Hochrieter and Schmidhuber [2], *"Learning to store information over extended period of time intervals via recurrent back propagation takes a very long time, mostly due to insufficient, decaying error back flow."*

While this drawback hinders the application of such sophisticated neural network designs, RNNs which utilize LSTM memory cells offer a solution for this problem as the memory cells provide forget and remember gates which prevent or lessen vanishing or exploding gradients. LSTM RNNs have been used successfully in many studies on involving time series data [3, 4, 5, 6, 7] and were chosen by this study to examine them as a solution to predicting aircraft engine vibration.

For many years, neural networks have strongly proven their prediction potential and applied in different many areas [8, 9, 10]. Various strategies and techniques exist to automatically generate the structure of neural networks and the most common used ones are based on evolutionary algorithms. These neuroevolution strategies examine the use of learning and evolution as concepts to improve the performance of neural networks. In traditional neuroevolution [11], an evolutionary algorithm is used to train a neural networks' connection weights with a fixed structure, but significant benefit has been demonstrated in using these techniques to both optimize and evolve connections and topologies [12, 13, 14], as weights are not the only key parameter for best performance of neural networks [15]. These strategies are of particular interest as determining the optimal structure for a neural network is still an open question. This particular work focuses on evolving the structure of LSTM neurons with an ant colony optimization [16] based algorithm.

1.1. Previous work

This study's ultimate goal is to explore the utilization of LSTM RNNs to predict future engine vibration in order to be used in a warning system to give indications for the problem before it occurs in order to

avoid or mitigate it. An initial work examined building viable Recurrent Neural Networks (RNN) using Long Short Term Memory (LSTM) neurons to predict aircraft engine vibrations [17]. To achieve this, three different LSTM RNNs architectures were examined to find which would provide better results. The three architectures varied in both complexity and depth of layers. The different networks were trained on time series flight data records obtained from a regional airline containing flights that suffered from excessive vibration. The structure of the LSTM RNNs used in this study is shown in Figure 6. After selecting an initial set of 15 relevant parameters, these LSTM RNNs were able to predict vibration values for 1, 5, 10, and 20 seconds in the future, with 2.84% 3.3%, 5.51% and 10.19% mean absolute error, respectively.

1.2. Ant Colony Optimization

Ant colony optimization (ACO) is a metaheuristic used to find approximate solutions to many combinatorial problems. It belongs to a family of bio inspired metaheuristics. ACO is a distributed approach using agents called artificial ants. These artificial ants resemble biological ants in that each ant is independent and communicates with other members of the colony through a chemical called pheromone. Ants randomly explore areas, however they tend to follow paths with pheromone, and upon finding food they mark their return path with more pheromone. Pheromones decay over time and paths with the most pheromone represent the most promising paths to food. The first algorithm of this type (the "Ant System" [18]) was designed for the traveling salesman problem, but failed to produce competitive results. However, subsequent research has shown this algorithm to be effective on this problem [19, 20, 21] and interest for the metaphor has launched many algorithms inspired by it in various fields, including continuous optimization problems [22, 23, 24, 25, 26, 27], and even training neural networks [28, 29, 30, 31].

1.3. Study's Contribution

This work improves the performance of the LSTM network architectures proposed in the previous work by optimizing LSTM cell structure. Since the contributions of the input parameters are not of the same magnitude to the problem (vibration), the weights of the neural network are adjusted through backpropagation. However, a fully connected neural network can pose additional training complexity and unwanted noise through some of the connections coming out of some of the inputs. Therefore, there is a need for a way to examine these connections and try to eliminate those which contribute to the prediction error. ACO has been chosen mainly because it has proven its effectiveness in evolving RNNs [16] for time series data prediction.

Using k-fold cross validation (k = 3), results demonstrate that the evolved LSTM architecture increase the best previous results' performance by 1.37% in predicting vibration 10 seconds in the future (reducing error from 6.35% to 5.01%), while at the same time only requiring nearly half of the connections (the number of weights was reduced from 21,770 to 11,650). The study also examined Nonlinear Output Error (NOE), a Nonlinear AutoRegression with Xogenous (NARX) input, and a Nonlinear Box-Jenkins (NBJ) recurrent neural networks which only reached error rates of 11.45%, 8.47% and 9.77%, respectively.

2. Related Work

2.1. Turbine Engine Vibration

According to A. V. Srinivasan [1]: *"The most common types of vibration problems that concern the designer of jet engines include (a) resonant vibration occurring at an integral order, i.e. multiple of rotation speed, and (b) flutter, an aeroelastic instability occurring generally as a nonintegral order vibration, having the potential to escalate, unless checked by any means available to the operator, into larger and larger stresses resulting in serious damage to the machine. The associated failures of engine blades are referred to as high cycle fatigue failures".* The *means available to the operator* in practical aviation operations are mainly the implementation of manufacturers' maintenance program which relies on reliability observations.

Any airplane maintenance program has four objectives: *i)* guaranteeing the inherit safety and reliability levels of the systems and subsystems, *ii)* restore those levels to their original levels if deviations occur, *iii)* gather information about design enhancement for systems and subsystems that showed deficiency in there expected reliability, and *iv)* accomplish these targets at the the most overall cost efficiency possible. To achieve these goals a maintenance program consists of checks performed over specific intervals to perform on-condition checks which might be visual inspections, test runs, or non-destructive tests performed on the systems' or

subsystems components. In addition to that, and as mentioned earlier, preventive maintenance is also part of the maintenance program where parts are replaced or overhauled based on reliability observations [32].

There are also other methods to mitigate the risk coming from excessive engine vibration using statistical and holistic computation methods. This is accomplished by monitoring the engine performance through its flight data history, which is logged as time series data, in order to forecast the excessive vibration occurrence. However, since these methods are not exact, reasonable safety factors should be considered. As discussed in the following sections, machine learning and artificial intelligence becomes the heart of such methods.

2.2. Time Series Data Prediction

From a statistical point of view, the main goal of prediction is to provide vital information for decision makers, economists, planners optimizers, industrialists and critical systems operators. There are two sides for prediction: the qualitative side and the quantitative side. The qualitative side utilizes methods known as the judgmental or subjective prediction methods which covers methods relaying on intuition, judgement or opinions of some kind of a referee as consumers, experts and/or supporting information. Qualitative methods are considered in cases when past data is not available. On the other hand, quantitative methods include univariate and multivariate methods. For many study cases related to different scientific and real life problems, the time series data are available on several dependent variables, and in such cases multivariate prediction methods are used [33].

The models in the time series predictions realm mainly falls in two categories: *a)* statistical prediction models which include, *e.g.*, the autoregressive (AR) model, the moving average (MA) model, and hybrid models that derive from them such as autoregressive moving average (ARMA), autoregressive integrated moving average (ARIMA), seasonal ARMIA (SARIMA) [34], vector autoregressive (VAR) models and *b)* artificial neural networks (ANN) prediction models. While successful studies have managed to achieve good results by using such methods and even reported results that out perform neural networks [35], the main draw back of the statistical methods is that they generally can not be applied to non-linear systems [34]. On the other hand, ANNs have shown good performance with such systems. There are also a third category which are the hybrid models. These models (hybrid) offer the benefits of both statistical and ANN models. Consequently, as the studied system involving prediction of aircraft engine vibration is complicated in its nature and is expected to be extremely non-linear, statistical and hybrid models are considered beyond the scope of the study.

Ranković *et al.* [36] discussed a neural network that exploited the concept behind the nonlinear autoregression but with exogenous recurrent inputs. Where Θ is the regressor, y is the target data, $\hat{y}$ is the output, e is the error, t is the time step, and n is the lag limit, the proposed neural networks in the study were the:

- *Nonlinear Finite Impulse Response (NFIR) model:*

$$\Theta(t) = (i_{t1}, i_{t2}, ..., i_{tn}) \tag{1}$$

- *Nonlinear AutoRegressive with eXogenous inputs (NARX) model:*

$$\Theta(t) = (i_{t1}, i_{t2}, \ldots, i_{tn}, y_{t-1}, y_{t-2}, \ldots, y_{t-n}) \tag{2}$$

- *Nonlinear AutoRegressive Moving Average with eXogenous inputs (NARMAX) model:*

$$\Theta(t) = (i_{t1}, i_{t2}, \ldots, i_{tn}, y_{t-1}, y_{t-2}, \ldots, y_{t-n}, e_{t-1}, e_{t-2}, \ldots, e_{t-n})) \tag{3}$$

- *Nonlinear Output Error (NOE) model:*

$$\Theta(t) = (i_{t1}, i_{t2}, \ldots, i_{tn}, \hat{y}_{t-1}, \hat{y}_{t-2}, \ldots, \hat{y}_{t-n}) \tag{4}$$

- *Nonlinear Box-Jenkins (NBJ) model:* which uses all four regressor types.

Although the the data of the study is not stationary, as the the data is dependent on flight control inputs, this study also examined using the NOE, NARX, and NBJ models as a baseline for comparison to the study's LSTM Architecture I and ant colony optimized architecture.

2.3. Time Series Prediction in Aviation

Some effort has been done using neural networks to classify engine abnormalities without doing analytical computation, *e.g.*, Alexandre Nairac *et al.* [37] have performed research to detect abnormalities in engine vibrations based on recorded data. To achieve that, the work used two modules. One of the modules uses the overall shape of the vibration curve to detect unusual vibration signatures. The second one reports sudden unexpected transitions in the signature curves. Their approach to detect defects is not to introduce examples of faulty engines to the neural network, rather, only examples of healthy engines are introduced to the neural networks in the training phase. This approach was taken to overcome the lack of existence of adequate faulty engine data, which was not enough for training. In this context, the paper introduces the term 'normality' to describe the behavior of normal engines and 'abnormality' to describe the behavior of faulty engines. Using statistical models, the faulty engines detection would be described as 'novelty' detection based on the deviation from the data distribution. The best results this work achieved was the prediction of faulty engines with 84% successful classifications.

David A. Clifton *et al.* [38] presented work for predicting abnormalities in engine vibration based on statistical analysis of vibration signatures. The paper presents two modes of prediction. One is ground-based (off-line), where prediction is done by run-by-run analysis to predict abnormalities based on previous engine runs. The success in this approach was predicting abnormalities two flights ahead. The other mode is a flight based-mode (online) in which detection is done either by sending reduced data to the ground-base or processing it onboard the aircraft. The paper mentions that they could successfully predict vibration events 2.5 hours in the future. However, this prediction is done after half an hour of flight data collection, which might be a critical time as well, as excess vibration may occur during this data collection time. The paper did not mention how much data was required to have a sound prediction.

2.3.1. RNN for Predicting Flight Parameters

Having an advantage over standard FFNNs[1], RNNs can deal with sequential input data, using their internal memory to process sequences of inputs and use previously stored information to aid in future predictions. This is done by feedback connections or by looping between neurons, which allows them to be of predicting more complex data [39].

This presented work is in part inspired by previous work on predicting flight parameters [40, 16]. Which first utilized evolutionary algorithms such as particle swarm optimization [41, 42] and differential evolution [43] to optimize the weights of the network [40], and then an ant colony optimization based algorithm to evolve different recurrent neural network structures [16]. The neural networks evolved with ant colony optimization predicted airspeed, altitude and pitch with a 63%, 97% and 120% improvement respectively over the previously best published results. The research used recurrent neural networks and applied an ant-colony optimization (ACO) algorithm [44, 45, 46], an optimization technique used in the beginning on discrete problems, mainly on the Traveling Salesman Problem [47]. Later it was used in continuous optimization problems [22, 23, 24, 25, 26, 27] including training neural networks [28, 29, 30, 31].

2.3.2. LSTM RNN

LSTM RNNs were first introduced by S. Hochrieter & J. Schmidhuber [4]. While the work by T. Desell *et al.* utilized non-gradient based evolutionary algorithms to optimize RNN weights, LSTM neurons provide a solution for the exploding/vanishing gradients problem by utilizing various gates, which allow backpropagation to be used in large RNNs (S. Hochrieter in 1991). This work has paved the way for many interesting projects.

Later, J. Schmidhuber *et al.* [48] emphasized the forget gate in the LSTM RNNs. The paper mentions that *"We identify a weakness of LSTM networks processing continual input streams that are not a priori segmented into subsequences with explicitly marked ends at which the network's internal state could be reset. Without resets, the state may grow indefinitely and eventually cause the network to break down. Our remedy is a novel, adaptive forget gate that enables an LSTM cell to learn to reset itself at appropriate times, thus releasing internal resources. We review illustrative benchmark problems on which standard LSTM outperforms*

[1] Feed Forward Neural Networks

other RNN algorithms. All algorithms (including LSTM) fail to solve continual versions of these problems. LSTM with forget gates, however, easily solves them, and in an elegant way." However, Felix A. Gers *et al.* [49] suggest that *"LSTM RNNs does not carry over to certain simpler time series prediction tasks solvable by time window approaches"*. The paper suggests to use LSTM when *"simpler traditional approaches fails"*.

LSTM RNNs have been used with strong performance in image recognition [50], audio visual emotion recognition [51], music composition [52] and other areas. Regarding time series prediction, for example, LSTM RNNs have been used for stock market forecasting [3] and forex market forecasting [7]. Also forecasting wind speeds [4, 5] for wind energy mills, and even predicting diagnoses for patients based on health records [6].

2.4. Evolutionary Optimization Methods

Several methods for evolving topologies along with weights have been searched and deployed. In [53, 53], NeuroEvolution of Augmenting Topologies (NEAT) has been developed. It is a genetic algorithm that evolves increasingly complex neural network topologies, while at the same time evolving the connection weights. Genes are tracked using historical markings with innovation numbers to perform crossover among different structures and enable efficient recombination. Innovation is protected through speciation and the population initially starts small without hidden layers and gradually grows through generations [54, 55, 56]. Experimentations have demonstrated that NEAT presents an efficient way for evolving neural networks for weights and topologies in parallel or separately. Its power resides in its ability to combine all the four main aspects discussed above and expand to complex solutions along the generation process. However NEAT still has some limitations when it comes evolving neural networks with weights or LSTM cells for time series prediction tasks as it has been claimed in [16].

2.5. RNN Regularization vs. ACO Optimization

Srivastava *et al.* have demonstrated the ability to utilize Dropout, a popular method for regularization of convolutional neural networks [57], to be applied as a regularizer for recurrent neural networks [58]. This work has shown strong results in reducing overfitting when utilizing large RNNs. As dropout randomly drops out connections during the forward pass of the backpropagation algorithm, it effectively trains the network over randomly sampled subnetworks of the fully connected architecture. As each forward path is a different randomly selected network, this forces the trained weights to become more robust and serves to reduce overfitting.

While this approach is highly successful for classification problems, such as those presented in Zaremba *et al.*'s work [58], which can easily be overfit – this work focuses on time series data prediction multiple readings in the future. In all tested architectures, the RNNs have not come close to overfitting on the training data, but rather the problem has been effectively training the network given the highly challenging prediction task. The ACO approach described in this work focuses on finding the best subset of connections to use in an RNN, which makes training them more efficient and effective – using a fixed sub-topology for an entire training process, as opposed to randomly dropping out connections in each forward pass, as done by dropout.

3. Methodology

This work utilizes the ACO method developed by Desell *et al.* to evolve the structure of LSTM cells, in part due to strong previous results and because it allows any method to be used to determine the optimal weights of connections. This is particularly important as it allows backpropagation to be used on a large scale LSTM RNN which is significantly more efficient than the non-gradient based evolutionary algorithms used in previous work.

3.1. Experimental Data

The flight data used consists of 76 different parameters recorded on the aircraft Flight Data Recorder (FDR), inclusive of the engine vibration parameters. During the data processing phase of the project, two efforts were done to identify the parameters that most contributed to the engine vibration.

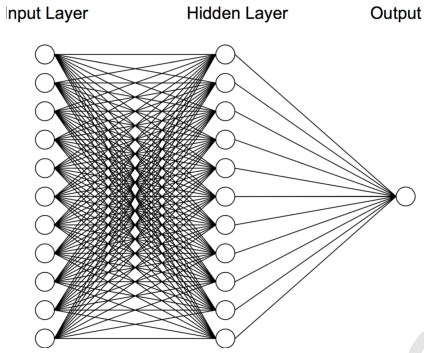


Figure 1: A one layer feed forward neural network.

3.1.1. Data Correlation Parameter Selection

Primarily, cross-correlation analysis [59] was exercised to find the potential parameters that highly contribute to vibration. Every parameter from each flight was cross-correlated to vibration then plotted to pick the highest correlated parameters. Cross correlation was calculated using the following Equation:

$$Cross\_Correlation = \sum_{\infty}^{\infty} x[a] \cdot vib[a] \tag{5}$$

Where the highest correlation was determined by calculating the area under the plotted curve for the normalized data. The top correlated parameters to vibration were:

1. Right InBoard Spoiler
2. Right OutBoard Spoiler
3. Left InBoard Spoiler
4. Left OutBoard Spoiler
5. Static Air Temperature
6. Pitch 2
7. Pitch
8. Slat Configuration
9. Main Landing Gear Lock Down Sensors
10. Flap Configuration

A one layer feed forward neural network was built as shown in Figure 1 to predict vibration given other parameters within the same second. However, the results were poor, with significant noise in the predictions. This imposed a question about the quality of the chosen parameters using this method and due to this, another method of parameter-selection was sought. A potential cause for such misleading cross-correlation chosen parameters was that some flight configuration parameters like spoilers/slats/flaps positions, pitch angle and main-landing-gear position do not change but few times during the flight, which can translate into high correlation with the vibration.

3.1.2. Aerodynamics/Turbo-machinery Parameter Selection

A second subset of the FDR parameters were then chosen based on the likelihood of their contribution to the vibration based on aerodynamics/turbo-machinery expert knowledge. Again, a one layer feed forward neural network with a structure similar to the one shown in Figure 1, except the number of input and hidden nodes was equal to the number of chosen parameters, was applied and these results were encouraging enough to use these parameters for predicting vibration in future.

Some parameters such as Inlet Guide Vans Configuration, Fuel Flow, Spoilers Configuration (this was preliminarily considered because of the special position of the engine mount), High Pressure Valve Configuration and Static Air Temperature were excluded because it was found that they generated more noise than positively contributing to the vibration prediction.

The final chosen parameters were:

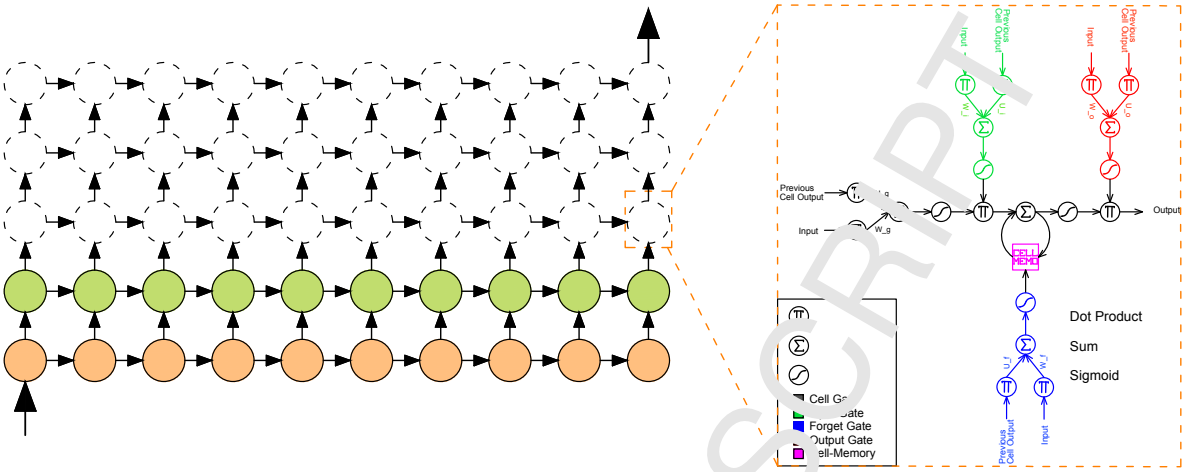


Figure 2: A generic overview of the design of a LSTM RNN.

1. Altitude [ALT]
2. Angle of Attack [AOA]
3. Bleed Pressure [BPRS]
4. Turbine Inlet Temperature [TIT]
5. Mach Number [M]
6. Primary Rotor/Shaft Rotation Speed [N1]
7. Secondary Rotor/Shaft Rotation Speed [N2]
8. Engine Oil pressure [EOP]
9. Engine Oil Quantity [EOQ]
10. Engine Oil Temperature [EOT]
11. Aircraft Roll [Roll]
12. Total Air Temperature [TAT]
13. Wind Direction [WDir]
14. Wind Speed [WSpd]
15. Engine Vibration [Vib]

3.2. Recurrent Neural Network Design

Three LSTM RNN architectures were designed to predict engine vibration 5 seconds, 10 seconds, and 20 seconds in the future. Each of the 15 selected FDR parameters is represented by a node in the inputs of the neural network and an additional node is used for a bias. Each neural network in the three designs consists of LSTM cells that receive both an initial input of flight data at some time in the past or the output from a cell in the lower layer, and the output of the previous cell in the same layer, as inputs (see Figure 2). Each cell has three gates to control the flow of information through the cell and accordingly, the output of the cell. Each cell also has a cell-memory which is the core of the LSTM RNN design. The cell-memory allows the flow of information from the previous states into the current predictions.

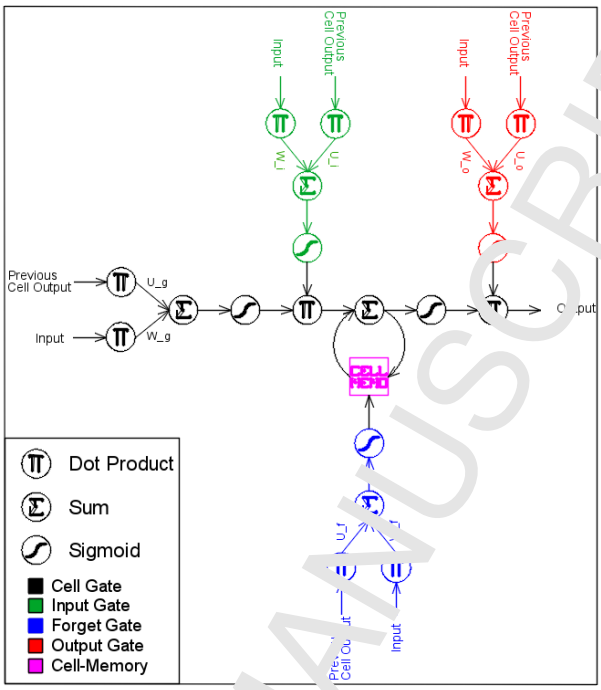
The gates that control the flow are shown in Figure 3. They are: *i)* the *input gate*, which controls how much information will flow from the inputs of the cell, *ii)* the *forget gate*, which controls how much information will flow from the cell-memory, and *iii)* the *output gate*, which controls how much information will flow out of the cell. This design allows the network to learn not only about the target values, but also about how to tune its controls to reach the target values.

All the utilized architectures follow the common LSTM RNN designs shown in Figure 2 and 3. However, there are two variations of this common design used in the utilized architectures, shown in Figures 4 and 5, with the difference being the number of inputs from the previous cell. Cells that take an initial number of inputs and output the same number of outputs are denoted by *'M1'* cells. As input nodes are needed to be reduced through the neural network, the design of the cells are different. Cells which perform a reduction on the inputs are denoted by *'M2'* cells.

3.2.1. LSTM RNN Forward Propagation Equations

The equations used in the forward propagation through the neural network are:

$$i_t = Sigmoid(w_i \bullet x_t + u_i \bullet a_{t-1} + bais_i) \tag{6}$$



Fig[illegible] LSTM cell design

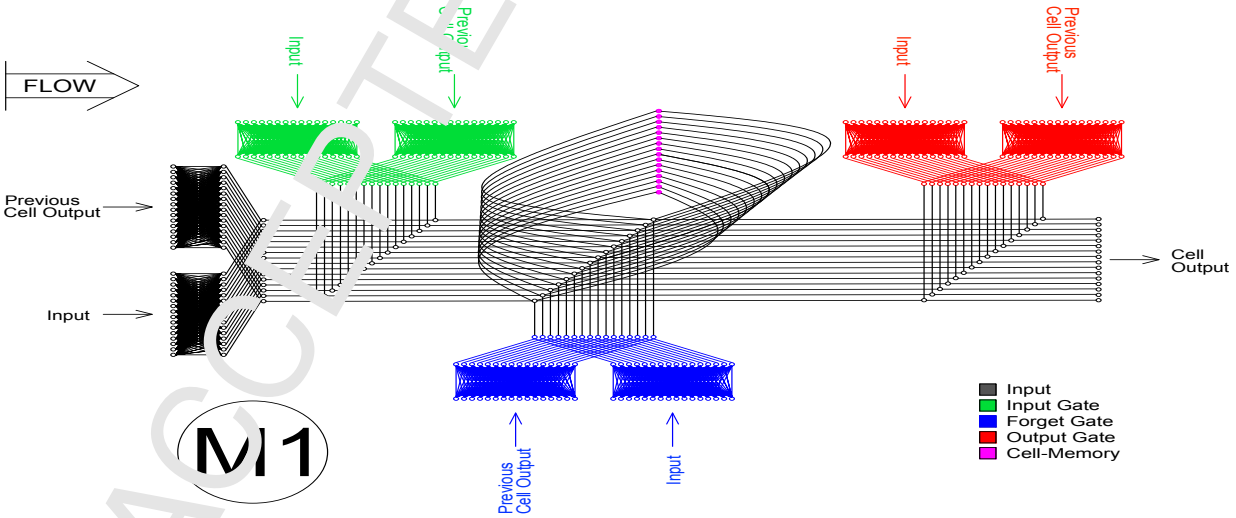


Figure 4: Level 1 LSTM cell design

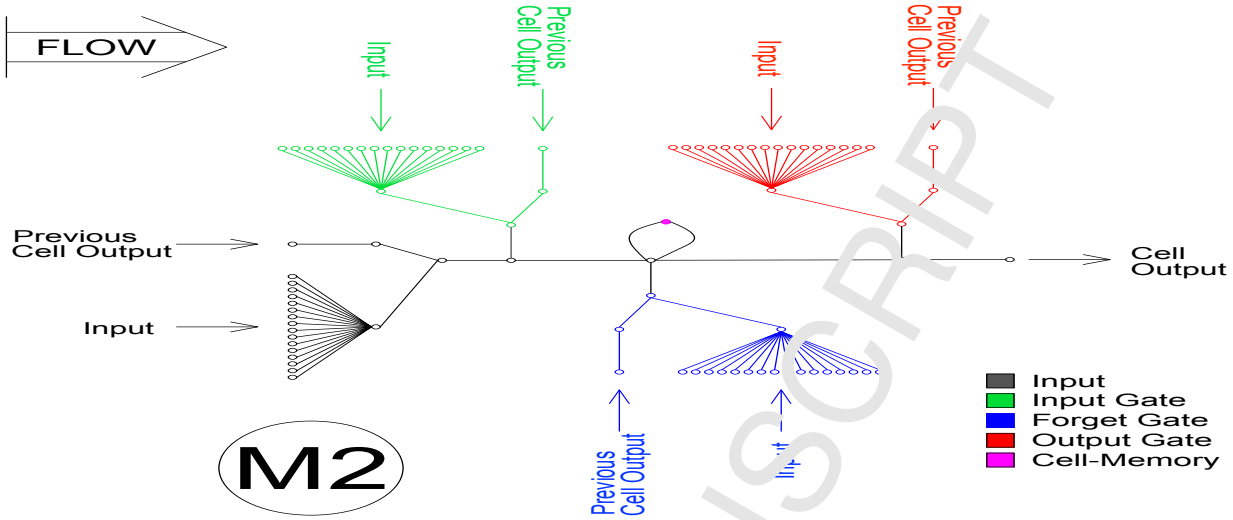


Figure 5: Level 2 LSTM cell design

$$f_t = Sigmoid(w_f \bullet x_t + u_f \bullet a_{t-1} + bais_f) \tag{7}$$

$$o_t = Sigmoid(w_o \bullet x_t + u_o \bullet a_{t-1} + bais_o) \tag{8}$$

$$g_t = Sigmoid(w_g \bullet x_t + u_g \bullet a_{t-1} + bais_g) \tag{9}$$

$$c_t = f_t \bullet c_{t-1} + i_t \bullet g_t \tag{10}$$

$$a_t = o_t \bullet Sigmoid(c_t) \tag{11}$$

where (see Figure 3):

i_t: input-gate output
f_t: forget-gate output
o_t: output-gate output
g_t: input's sigmoid
c_t: cell-memory output
w_i: weights associated with input and input-gate
u_i: weights associated with previous output and input-gate
w_f: weights associated with input and forget-gate
u_f: weights associated with previous output and forget-gate
w_o: weights associated with input and output-gate
u_o: weights associated with previous output and the output-gate
w_g: weights associated with the cell input
u_g: weights associated with previous output and the cell input

and the formula of the sigmoid function is:

$$Sigmoid(\alpha) = \frac{1}{1 + e^{-\alpha}} \tag{12}$$

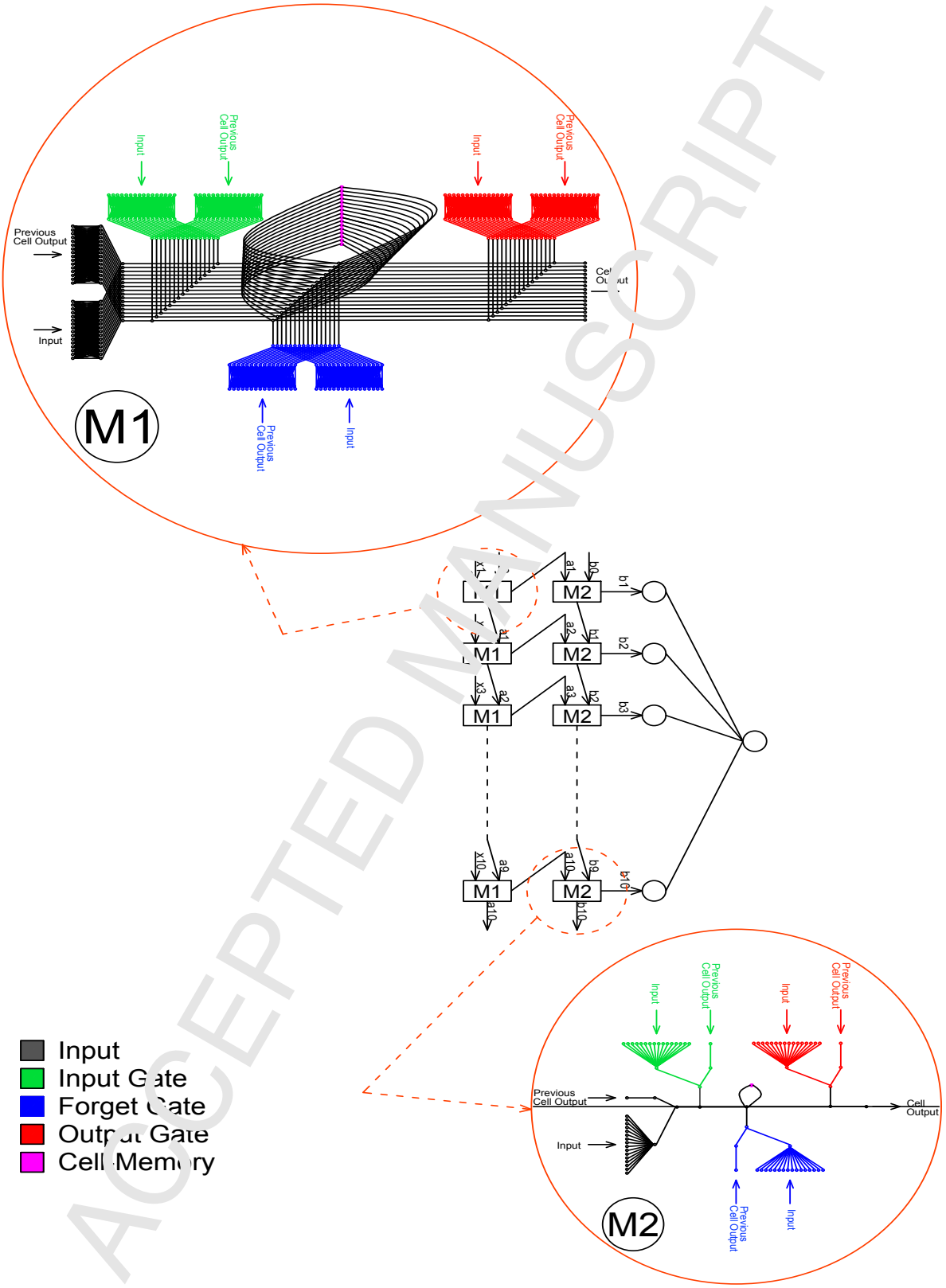


Figure 6: Neural network structure

3.3. LSTM RNN Architectures

The three architectures are as follows, with the dimensions of the weights of these architectures shown in Table 1 and the total number of weights shown in Table 2.

Architecture I

As shown in Figure 7a, the first level of the architecture takes inputs from ten time series (the current time instant and the past nine). It then feeds the second level of the neural network with the output of the first level. The output of the first level of the neural network is considered the first hidden layer. The second level of the neural network then reduces the number of nodes fed to it from 16 nodes (15 input nodes + bias) per cell to only one node per cell. The output of the second level of the neural network is considered the second hidden layer. Finally, the output of the second level of the neural network would be only 10 nodes, a node from each cell. These nodes are fed to a final neuron in the third level to compute the output of the whole network.

The dimensions of the weights matrices and vectors of this architecture are shown in Table 1. The total number of weights are is shown in Table 2. Figures 8 and 9 provide an overview of architecture I, as it has a large number of connections (21,170). Figure 8 shows the overall design of how the LSTM cells are connected, and then Figure 9 displays all the connections within a single time step of the full LSTM RNN. As a whole, there are 10 different instances of Figure 8, each connected as specified in Figure 9.

Architecture II

As shown in Figure 7b, this architecture is almost the same as the previous one except that it does not have the third level. Instead, the output of the second level is averaged to compute the output of the whole network.

The dimensions of the weights matrices and vectors of this architecture are shown in Table 1. The total number of weights are shown in Table 2.

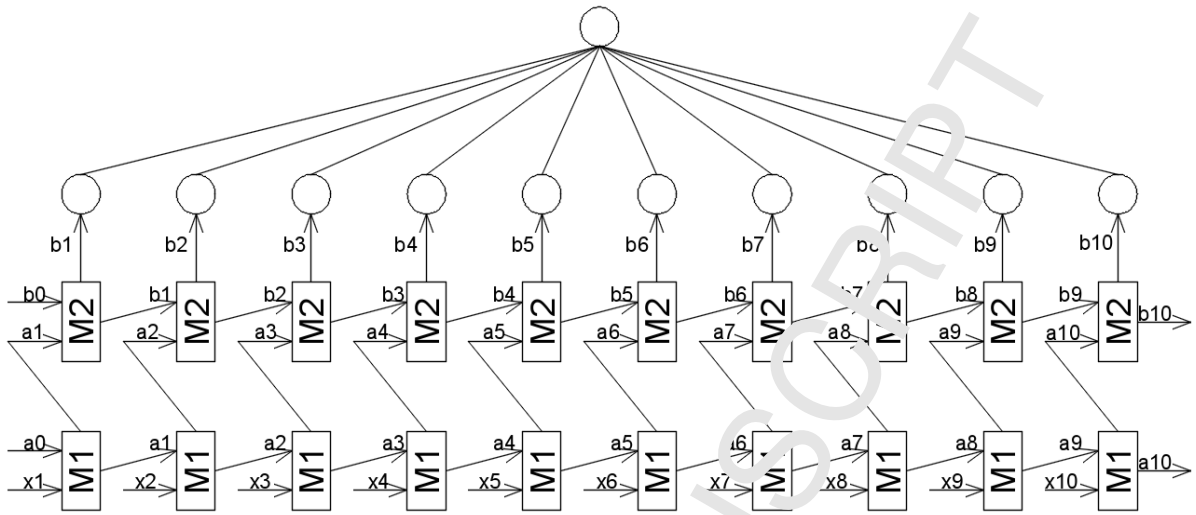
Architecture III

Figure 7c presents a deeper neural network architecture. In this design, the neural network takes inputs from twenty time series (the current time instant and the past nineteen) as the first level. It feeds the second level of the neural network with the output from the first level. The second level does the same procedure as first level giving a chance for more abstract decision making. The output of the second level of the neural network is considered the first hidden layer and the output of the second level is considered the second hidden layer. The third level of the neural network then reduces the number of nodes fed to it from 16 nodes (15 input nodes + bias) per cell to only one node per cell. The output of the third level of the neural network is considered the third hidden layer. Finally, the output of the third level of the neural network is twenty nodes, a node from each cell. These nodes are fed to a final neuron in the fourth level to compute the output of the whole network.

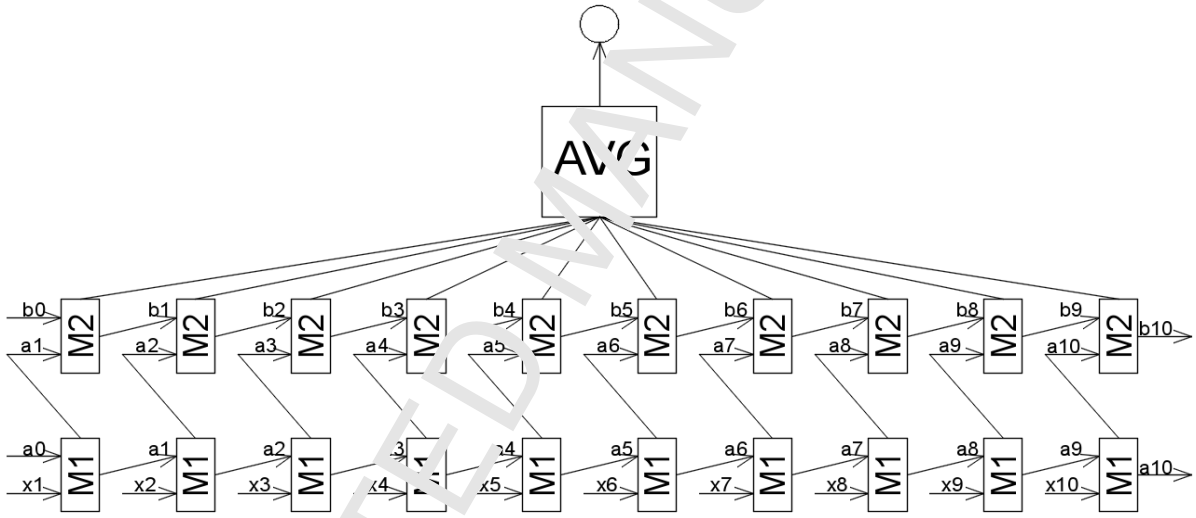
The Dimensions of the weights matrices and vectors of this architecture are shown in Table 1. The total number of weights are shown in Table 2.

3.4. Forward Propagation

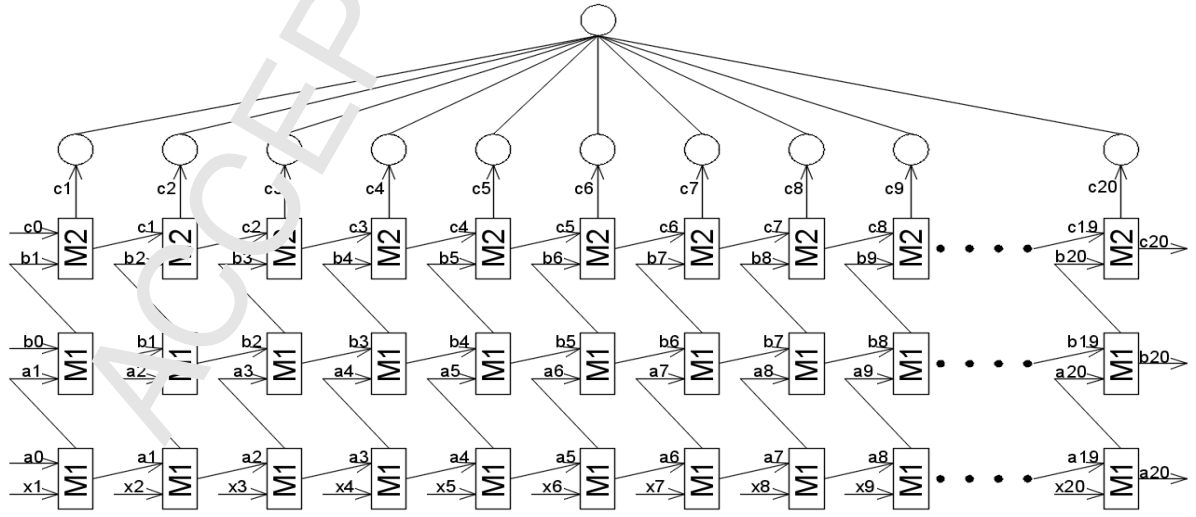
The following is a general description for the forward propagation path. This example uses Architecture I as an example but similar steps are taken in the other architectures with minor changes apparent in their diagrams. With Figure 7a presenting an overview of the structure of the whole network and considering Figure 4 as an overview of the structure of the cells in *Level 1* (M1) and Figure 5 as an overview of the structure of the cells in *Level 2* (M2) – the input at each iteration consists of 10 seconds of time series data of the 15 input parameters and 1 bias (*Input* in Figure 4) in one vector (x_t in Figure 7a) and the output of the previous cell (*Previous Cell Output* in Figure 4) in another vector (a_{t-1} in Figure 7a). Each second of time series input is fed to the corresponding cell (*i.e.*, the first seconds' 15 parameters and 1 bias are fed to first cell, the second seconds' 15 parameters and 1 bias are fed to second cell, ...) into the *cell gate* (shown in black color), *input gate* (shown in green color), *forget gate* (shown in blue color) and the *output gate* (shown in red color). If the gates (*input gate*, *forget gate* and, *output gate*) are seen as valves that control how much of the data flow through it, the outputs of these gates (i_t, f_t and, o_t) are considered as how much these valves are opened or closed.



(a) Architecture I



(b) Architecture II



(c) Architecture III

Figure 7: Used LSTM RNNs Architectures

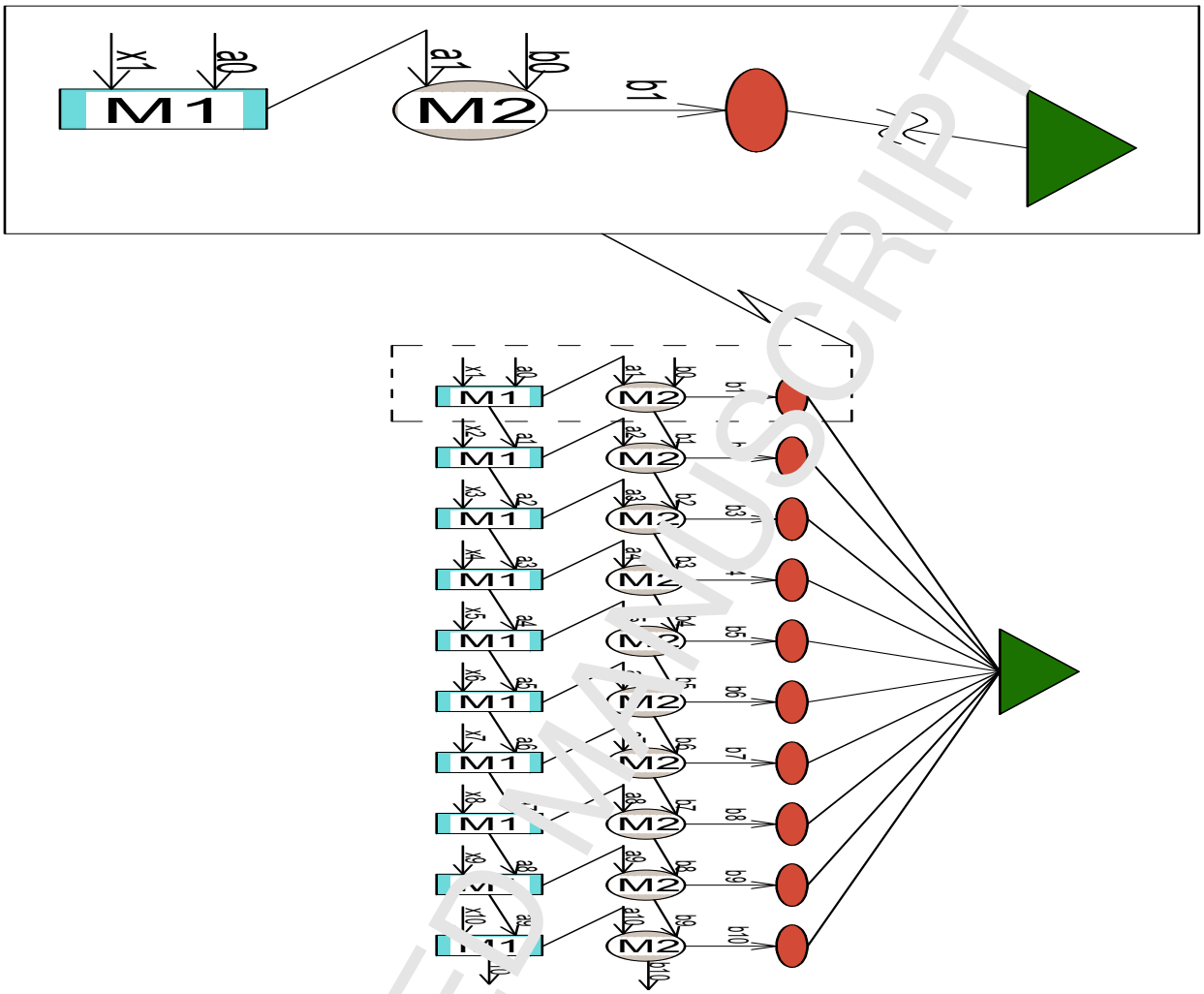


Figure 8: One time step of Architecture I

Table 1: Architectures Weights-Matrices Dimensions

Architecture I

	w_i	u_i	w_f	u_f	w_o	u_o	w_g	u_g
Level 1	16×16	16×16	16×16	16×16	16×16	16×16	16×16	16×16
Level 2	16×1	1×1	16×1	1×1	16×1	1×1	16×1	1×1
Level 3	16×1							

Architecture II

	w_i	u_i	w_f	u_f	w_o	u_o	w_g	u_g
Level 1	16×16	16×16	16×16	16×16	16×16	16×16	16×16	16×16
Level 2	16×1	1×1	16×1	1×1	16×1	1×1	16×1	1×1

Architecture III

	w_i	u_i	w_f	u_f	w_o	u_o	w_g	u_g
Level 1	16×16	16×16	16×16	16×16	16×16	16×16	16×16	16×16
Level 2	16×16	16×16	16×16	16×16	16×16	16×16	16×16	16×16
Level 3	16×1	1×1	16×1	1×1	16×1	1×1	16×1	1×1
Level 4	16×1							

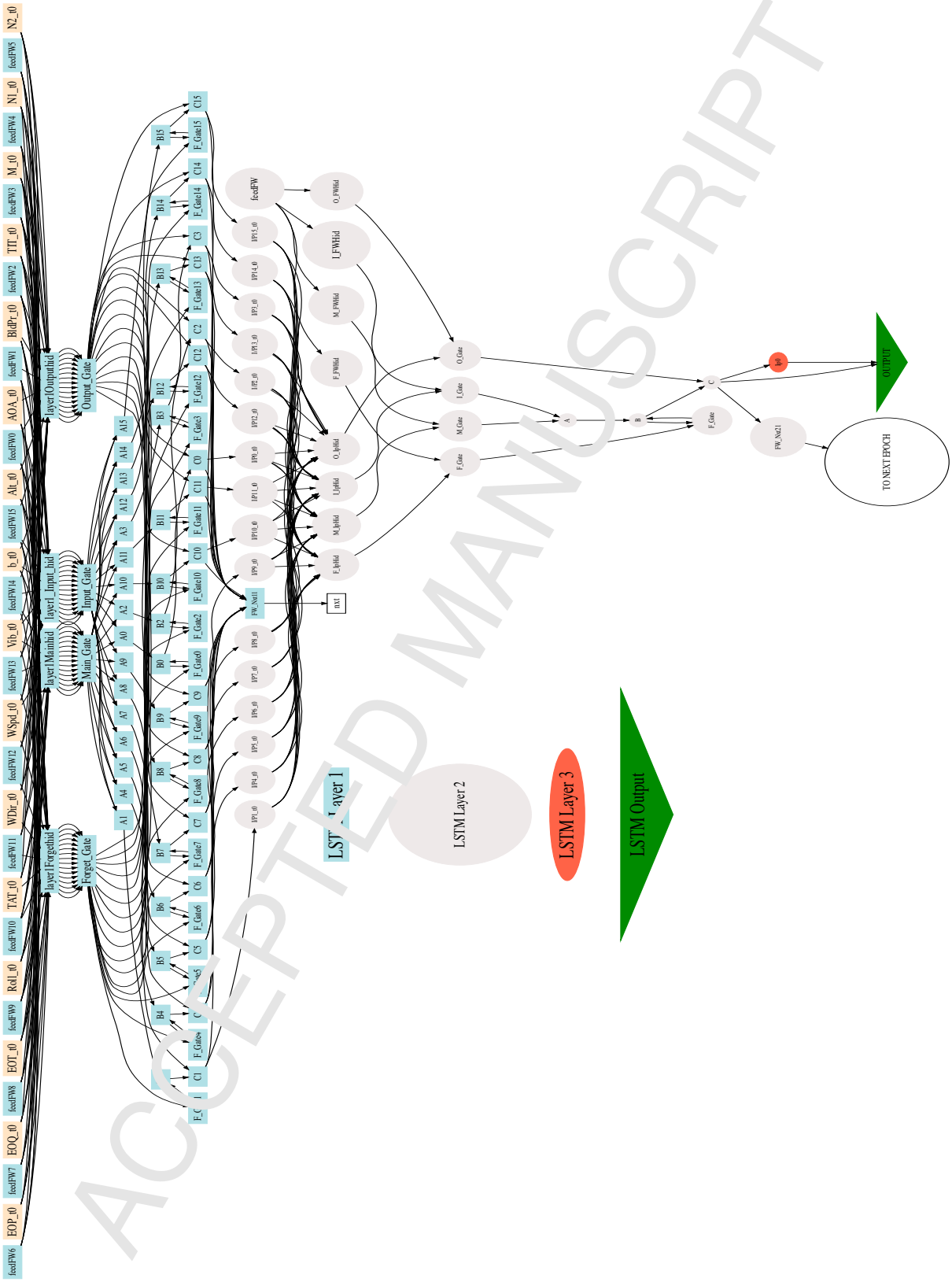


Figure 9: One time step of the Architecure I Full Structure

Table 2: Architectures Weights Matrices' Total Elements

Architecture I	Architecture II	Architecture III
21,170	21,160	83,290

First, at the *cell gate*, x_t is dot multiplied by its weights matrix w_g and a_{t-1} is dot multiplied by its weights matrix u_g. The output vectors are summed and an activation function is applied to it as in Equation 9. The output is called g_t.

Second, at the *input gate*, x_t is dot multiplied by its weights matrix w_i and a_{t-1} is dot multiplied by its weights matrix u_i. The output vectors are summed and an activation function is applied to it as in Equation 6. The output is called i_t.

Third, at the *forget gate*, x_t is dot multiplied by its weights matrix w_f and a_{t-1} is dot multiplied by its weights matrix u_f. The output vectors are summed and an activation function is applied to it as in Equation 7. It controls how much of the *cell memory* Figure 7a (saved from previous time-step) should pass. The output is called f_t.

Fourth, at the *output gate*, x_t is dot multiplied by its weights matrix w_o and a_{t-1} is dot multiplied by its weights matrix u_o. The output vectors are summed and an activation function is applied to it as in Equation 8. The output is called o_t.

Fifth, the contribution of the cell input *Input* g_t and *cell memory* c_{t-1} is decided in Equation 10 by dot multiplying them by f_t and i_t respectively. The output of this step is the new *cell memory* c_t.

Sixth, cell output is also regulated by the output gate (valve). This is done by applying the sigmoid function to the *cell memory* c_t and dot multiplying it by o_t as shown in Equation 11. The output of this step is the final output of the cell at the current time-step a_t. a_t is fed to the next cell in the same level and also fed to the cell in the above level as an *Input* a_t.

The same procedure is applied at *Level 2* but with different weight vectors and different dimensions. Weights at *Level 2* have smaller dimensions to reduce their input detentions from vectors with 16 dimensions to vectors with one dimension. The output from *Level 2* is a one dimensional vector from each cell of the 10 cells in *Level 2*. These vectors are fed as one 10 dimensional vector to a simple neuron collection shown in Figure 7a at *Level 3* to be dot multiplied by a weight vector to reduce the vector to a single scalar value: the final output of the network at the time-step.

4. Evolving LSTM RNN Cells using Ant Colony Optimization

Although the results from Architecture I are promising, there is still room for further optimization in that the network may have excessive connections which confound accurate predictions and that the structure could be further optimized. A particular concern was that some connections could cause noise in the obtained results and ultimately would drift the results from their most optimum values, as this had been shown in the initial one layer feed forward neural networks with certain input parameters. The goal of using the ant colony optimization strategy is to evolve the structure of the LSTM cells, encouraging more diverse networks and selecting the topologies that give the best performance.

The ACO algorithm operates on the fully connected inputs to the M1 and M2 cells, as shown in Figures 4 and 5. Each M1 cell has eight 16x16 input gates, four of which take the input from the previous cell in the same layer, and four of which take the input from the time series or the cell in the lower layer. Each M2 cell has eight 8x1 input gates, four of which receive input from the previous cell in the same layer and four from the cell in the lower layer.

The algorithm begins with a fully connected gate that will be used by the ants each time to generate new paths for new network designs. Paths are selected by the ants based on pheromones – each connection in the network has a pheromone value that determine its probability to be chosen as a path. Given a number of ants, each one will select one path from the fully connected network. All the paths selected from all the ants are then collected, duplicated paths are removed and a design network is generated based on the new cell topology. Figure 10 shows an example on an M1 cell, assuming four ants choosing their paths on an input gate to an M1 cell, which generates a subgraph from the potentially fully connected input gate. The same

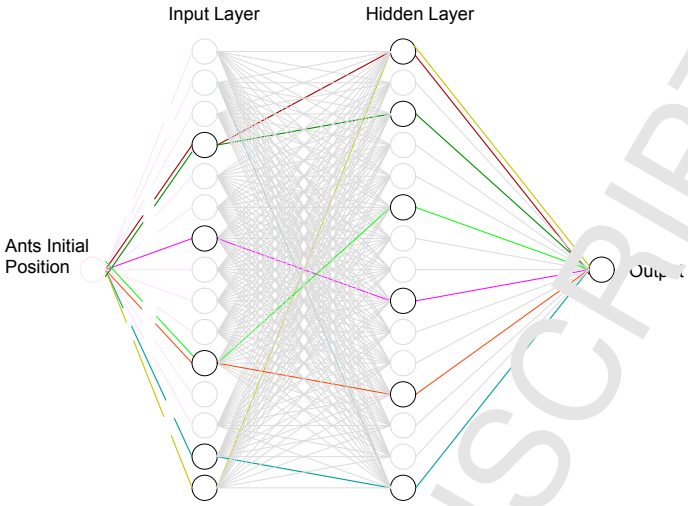


Figure 10: Schematic of Neural Network Structure after AOC

ACO generated topology is used for each of these 8 input gates. Figure 20a provides an example of the best found ACO optimized M1 cell.

In detail, the paths generated by ACO are used in the connections between the *"Input"* and the hidden layer neurons that follow it, and the *"Previous Cell Output"* and the hidden layer neurons that follow it. The connections between the *"Input"* and the hidden layer neurons that follow it are shown in the first level cells' (Figure 4 "M1") in BLACK color, GREEN color, BLUE color, and RED color at the gates of the cell. Once a hidden node in first level cell is reached by an ant, the connection between this node and the output node shown in second level cells' Figure 5 "M2" in BLACK color, GREEN color, BLUE color, and RED color, will automatically be part of the evolved mesh because the ant will not have any other option to reach the output node except through that single connection.

The same generated mesh is used at all the gates: Main Gate, Input Gate, Forget Gate, and Output Gate at the "M1"cells and "M2" cells at all the time-steps in the LSTM RNN Architecture I as shown in Figure 7a. In other words, regardless the LSTM RNN time-step, whenever there is a transition without data reduction: the first set of connections in the generated mesh is used, and whenever there is a transition with data reduction: the second set of connections in the generated mesh is used.

4.1. Distributed ACO Optimization

Evolving large LSTM RNNs is a computationally expensive process. Even training a single LSTM RNN is extremely time consuming (approximately 8.5-9 hours to train one architecture), and applying the ACO algorithm requires running the training process on each evolved topology. This significantly raises the computational requirements in time and resources necessary to process and evolve better LSTM networks. For that reason, the ant colony algorithm was parallelized using the message passing interface (MPI) for Python [60] to allow for it to be run utilizing high performance computing resources.

The distributed algorithm utilizes an asynchronous master worker approach, which has been shown to provide performance and scalability over iterative approaches in evolutionary algorithms [61, 62]. This approach provides an additional benefit in that it is automatically load balanced – workers request and receive new LSTM RNNs when they have completed training previous ones, without blocking on results from other workers. The master process can generate a new LSTM RNN to be trained from whatever is currently present in its population.

The algorithm is defined in Algorithm 1. In detail, the algorithm beings with the master process generating an initial set of network designs randomly (given a user defined number of ants), and sending these to the worker processes. When the worker receives a network design, it creates an LSTM RNN architecture by creating the LSTM cells with the according input gates and cell memory. The generated structure is then

trained on different flight data records using the backpropagation algorithm and the resulting fitness (test error) is evaluated and sent back along with the LSTM cell paths to the master process.

The master process then compares the fitness of the evaluated network to the other results in the population, inserts it into the population, and will reward the paths of the best performing networks by increasing the pheromones by 15% of their original value if it was found that the result was better than the best in the population. However, the pheromones values are not allowed to exceed a fixed threshold of 20. The networks that did not out perform the best in the population are penalized by reducing the pheromones along their paths by 15%. To control the pheromones values, all paths' pheromones are reduced every 100 iterations by 10%.

5. Implementation

5.1. Programming Language

Python's Theano Library [63] was used to implement the neural networks. It was chosen due to four major advantages: *i)* it will compile the most, if not all, of functions coded using it to C and CUDA providing fast performance, *ii)* it will perform the weights updates for backpropagation with minimal overhead, *iii)* Theano can compute the gradients of the error (cost function output) with respect to the weights, saving significant effort and time needed to manually derive the gradients, coding and debugging them (which is particularly challenging in regards to LSTM neurons), and finally, *iv)* it can utilize GPUs for further increased performance.

5.2. Data Processing

The flight data parameters used were normalized between 0 and 1. The sigmoid function was used as an activation function over all the gates and inputs/outputs. The ArcTan activation function was tested on the data, however it gave distorted results and sigmoid function provided significantly better performance.

5.3. Machine Specifications

The algorithm was implemented in Python using MPI for Python [60] and was run on the University of North Dakota's high performance computing cluster. The cluster is running the Red Hat Enterprise Linux (RHEL) 7.2 operating system with 31 nodes, each with 8 cores for 248 in total, 64GBs RAM per node for a total 1948 GB, and it is using InfiniBand 10 gigabit (GB) for interconnect. The same number of epochs (575) were used before to train the LSTM Network as in previous work for comparison purposes.

5.4. Using GPUs for LSTM RNN Training

The neural networks' weight matrices for a LSTM cell are repeated at a given time-step at a given layer. Thus, the computational cost increases if the output if these gates is computed separately, one gate at a time, as the data input/output consumes CPU cycles. This case is also obvious if a GPU is utilized for high performance computing as the cost of sending data forward and backward between the CPU (host) and GPU (device). For that, the input of a cell at a given layer is dot multiplied by a matrix that holds all of the gates weights concatenated one after the other. Then, the outputs; g Equation 9, i Equation 6, f Equation 7 and, o Equation 8, can be extracted from the dot product output matrix. Equation 13 is an example of combining (concatenating) the weights matrices for the LSTM cells' gates of level one in Architecture I. By this, all weights are transferred between the CPU and the GPU as one data structure, which would theoretically boost the performance.

These measures were followed when using the Theano library for GPU computing to manage the GPU threads, blocks and grids as well as the data transfer between the CPU and GPU. However, the performance was slower than compared to the pure CPU version. For Architecture I as an example, one iteration through the network during the learning process, it took the GPU version more than twenty minutes while it took slightly more than two minutes for the pure CPU version.

A further effort was made to overcome the data transfer penalty between the CPU and the GPU. The whole input data set was sent to the GPU as one data structure to avoid the data transfer through the iterations at every time series in the data and to perform those iterations on the GPU. Unfortunately, this

Algorithm 1 Ant Colony Algorithm

```
1: global variables
2:     ▷ A user specified number of ants.
3:     N_ANTS
4:     ▷ A user specified maximum number of pheromones.
5:     N_PHEROMONES
6:     ▷ The number of input parameters (15) + 1 for bias.
7:     N_INPUTS ← 16
8: end global variables
9: function GENERATE_PATHS
10:    paths = new Paths
11:    ▷ path arrays all initialized to 0. 1 indicates a connection.
12:    paths.input = array[16]
13:    paths.m1 = array[16][16]
14:    paths.m2 = array[16]
15:    for ant ← 1 ... n_ants do
16:        ▷ select input path probabilistically according to pheromones
17:        pheromone_sum ← sum(pheromones.input)
18:        r ← uniform_random(0, pheromone_sum − 1)
19:        input_path ← 0
20:        while r > 0 do:
21:            if r < pheromones.input[input_path] then
22:                paths.input_paths[input_path] ← 1
23:                break
24:            else
25:                r ← r − pheromones.input[input_path]
26:                input_path ← input_path + 1
27:        ▷ select hidden path probabilistically according to pheromones
28:        pheromone_sum ← sum(pheromones.m1[input_path])
29:        r ← uniform_random(0, pheromone_sum − 1)
30:        hidden_path ← 0
31:        while r > 0 do
32:            if r < pheromones.m1[input_path][hidden_path] then
33:                paths.m1_paths[input_path][hidden_path] ← 1
34:                paths.m2_paths[hidden_path] ← 1
35:                break
36:            else
37:                r ← r − pheromones.m1[input_path][hidden_path]
38:                hidden_path ← hidden_path + 1
       return paths
39: function UPDATE_PHEROMONES(pheromones, paths, action)
40:    for i ← 1 ... paths.input.length do
41:        if paths.input[i] = 1 then
42:            if action = REWARD then
43:                pheromones.input[i] ← min(pheromones.input[i] ∗ 1.15, MAX_PHEROMONE)
44:            else if action = PENALIZE then
45:                pheromones.input[i] ← pheromones.input[i] ∗ 0.85
46:            else if action = DEGRADE then
47:                pheromones.input[i] ← pheromones.input[i] ∗ 0.9
48:    for i ← 1 ... paths.m1.length do
49:        for j ← 1 ... paths.m1[i].length do
50:            if paths.m1[i][j] = 1 then
51:                if action = REWARD then
52:                    pheromones.m1[i][j] ← min(pheromones.m1[i][j] ∗ 1.15, MAX_PHEROMONE)
53:                else if action = PENALIZE then
54:                    pheromones.m1[i][j] ← pheromones.m1[i][j] ∗ 0.85
55:                else if action = DEGRADE then
56:                    pheromones.m1[i][j] ← pheromones.m1[i][j] ∗ 0.9
```

```
57: procedure MASTER
58:     ▷ pheromones all initialized to 1
59:     pheromones = new Pheromones
60:     pheromones.input ← array[16]
61:     pheromones.m1 ← array[16][16]
62:     pheromones.m2 ← array[16]
63:     population = List()
64:     iterations = 0
65:     repeat
66:         worker, message ← get_next_message()
67:         if message is request_paths then return generate_paths()
68:         else if message is report_fitness then
69:             fitness, paths ← message.get_arguments()
70:             rank ← population.inorder_insert( {fitness, paths} )
71:             if rank = 0 then
72:                 iterations ← iterations + 1
73:                 if fitness > max_fitness then
74:                     max_fitness ← fitness
75:                     update_pheromones(pheromones, paths, REWARD)
76:                 else
77:                     update_pheromones(pheromones, paths, PENALIZE)
78:                 if iterations  mod  100 = 0 then
79:                     update_pheromones(pheromones, paths, DEGRADE)
80:     until finished
81: procedure WORKER
82:     repeat
83:         paths ← master.request_paths()
84:         fitness ← LSTM_RNN(paths).backpropagate()
85:         master.report_fitness(fitness, paths)
86:     until finished
```

also did not help with the performance. Ultimately, a conclusion was reached that the subject matrices are not large enough to overcome the data transfer overhead. Further study is required to determine if it is possible to achieve good performance for these types of LSTM RNNs on GPUs.

$$\begin{bmatrix} w_{g_{1,1}} & w_{g_{1,2}} & w_{g_{1,3}} & \cdots & w_{g_{1,16}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{g_{16,1}} & w_{g_{16,2}} & w_{g_{16,3}} & \cdots & w_{g_{16,16}} \\ w_{i_1} & w_{i_2} & w_{i_{1,3}} & \cdots & w_{i_{1,16}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{i_{16,1}} & w_{i_{16,2}} & w_{i_{16,3}} & \cdots & w_{i_{16,16}} \\ w_{f_1} & w_{f_{1,2}} & w_{f_{1,3}} & \cdots & w_{f_{1,16}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{f_{16,1}} & w_{f_{16,2}} & w_{f_{16,3}} & \cdots & w_{f_{16,16}} \\ w_{o_{1,1}} & w_{o_{1,2}} & w_{g_{1,3}} & \cdots & w_{o_{1,16}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{o_{16,1}} & w_{o_{16,2}} & w_{o_{16,3}} & \cdots & w_{o_{16,16}} \end{bmatrix} \odot \begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \\ \vdots \\ x_{16} \end{bmatrix} = \begin{bmatrix} out_{g_1} \\ \vdots \\ out_{g_{16}} \\ out_{i_1} \\ \vdots \\ out_{i_{16}} \\ out_{f_1} \\ \vdots \\ out_{f_{16}} \\ out_{o_1} \\ \vdots \\ out_{o_{16}} \end{bmatrix} \tag{13}$$

5.5. Comparison to Traditional Methods

As mentioned in Subsection 2.2, the NOE, NARX, and NBJ models were implemented as baseline comparison methods. These traditional models are dynamical systems can experience limitations which reduce their stability and ability to make most effective use of embedded memory. In particular, they can suffer from vanishing and exploding gradients [64, 2], especially when using the back propagation through time algorithm [65] on long time series such as the vibration data used in this work.

It should further be noted that the purpose of this work is predict values multiple time steps into the future, which is not possible for the NBJ model, as it the actual value to be predicted along with the error between the prediction at that value to be fed back into the RNN at the next iteration. If this model is being used online to predict data 5, 10 or 20 seconds in the future, the output and error values will not be

known for an additional 5, 10 or 20 time steps (given readings every second) until that time actually occurs. However, as the data used in this study has already been collected, we still evaluated these models in an offline manner where this future knowledge can be known for sake of comparison.

5.5.1. Nonlinear Output Error (NOE) Inputs Neural Network:

The structure of the NOE network is depicted in Figure 11. The actual vibration values are fed as an inputs along with the current instance parameters and lag inputs. To make the model more comparable to the architectures used in this study, the parameters fed are the same used in the proposed LSTM RNN architectures to predict the vibration value in 10 seconds in the future (*i.e.*, they utilize the previous 10 seconds of input data, instead of just the current input data. The NOE does not have actual recurrent inputs, as it instead includes the actual prediction value as input instead. The vibration has been included as an input parameter in all models utilized, so the NOE model is no different than a traditional feed forward network.

5.5.2. Nonlinear AutoRegression with eXogenous (NARX) Inputs Neural Network:

This network, has been updated in a similar way to the NOE network. The previous 10 seconds of input data are utilized, and the previous 10 output values are fed to the network as recurrent inputs. Traditionally in the NARX model the weights for recurrent connects are fixed constants [66], and therefore their corresponding inputs are not considered in the gradient calculations and these weights are not updated in the training epochs. This was experimented on the data and the NARX network depicted in Figure 12 was used. However, the output of the cost functions in the training iterations of this implementation froze at a constant value, indicative of a case of vanishing gradients. Accordingly, the study allowed for the recurrent weights to be considered in the gradient calculations in order to update the weights with respect to the cost function output.

5.5.3. Nonlinear Box-Jenkins (NBJ) Inputs Neural Network:

The structure of the NBJ is depicted in Figure 13. As previously noted, this network is not feasible for prediction past one time step in the future in an online manner, as it requires the actual prediction value and error between it and the predicted value to be fed back into the network. However, as this work dealt with offline data, the actual future vibration values, error, and the output were all fed to the network along with the current instance parameters and lag inputs. As in the other networks, the values for the previous 10 time steps were also utilized.

5.6. K-Fold Cross Validation

The 57 flights in the data were divided into 10 groups. Seven groups consisted of 6 flights, and the other three consisted of 5 flights. The groups where used to cross validate the results by running nine of the groups as training data set and use the tenth as a testing set. Then, one of the groups in the training set switch places with the group in the testing set and then another run is executed. By doing so, the study had ten runs to cross validate the results and compare them statistically.

6. Results

6.1. Error Function

For all the networks studied in this work, Mean Squared Error (MSE) (shown in Equation 14) was used as an error measure for training, as it provides a smoother optimization surface for backpropagation than mean average error. Mean Absolute Error (MAE) (shown in Equation 15) was used as a final measure of accuracy for the three architectures, as because the parameters were normalized between 0 and 1, the MAE is also the percentage error.

$$Error = \frac{0.5 \times \sum(Actual\,Vib - Predicted\,Vib)^2}{Testing\,Seconds} \tag{14}$$

$$Error = \frac{\sum[ABS(Actual\,Vib - Predicted\,Vib)]}{Testing\,Seconds} \tag{15}$$

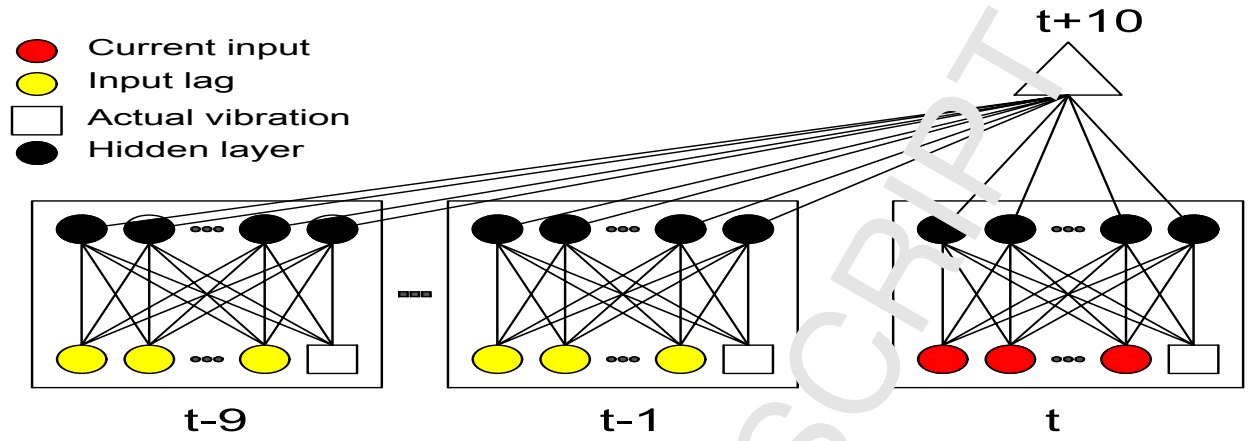


Figure 11: Nonlinear Output Error inputs neural network. This network was updated to utilize 10 seconds of input data.

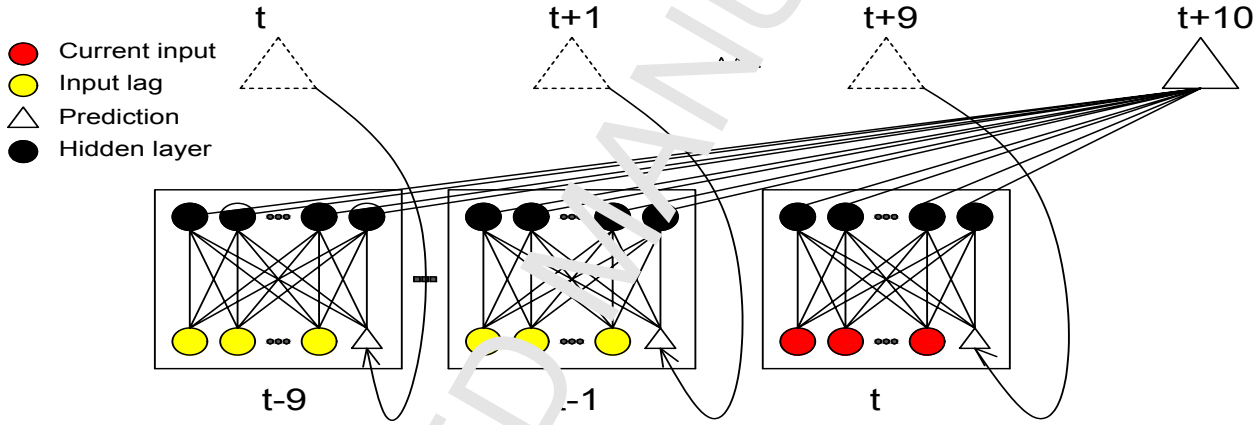


Figure 12: Nonlinear AutoRegressive with exogenous inputs neural network. This network was updated to utilize 10 seconds of input data, along with the previous 10 predicted output values.

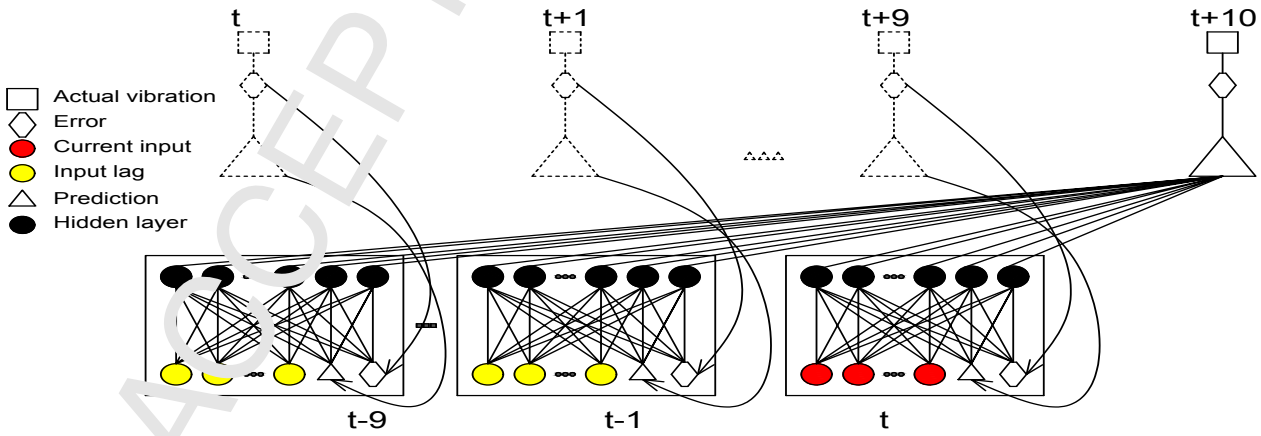
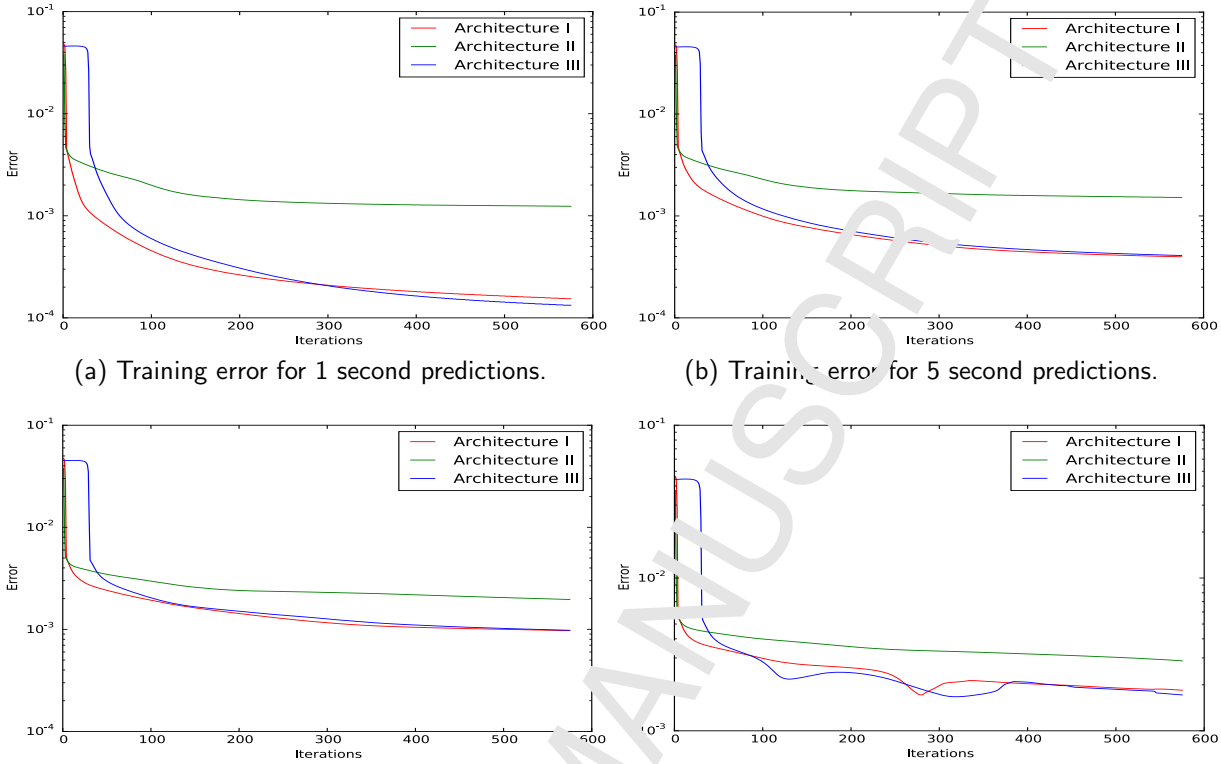


Figure 13: Nonlinear Box-Jenkins inputs neural network. This network was updated to utilize 10 seconds of input data, along with the future output and error values. Due to requiring future knowledfge, it is not possible to utilize this network in an online fashion.



(a) Training error for 1 second predictions.

(b) Training error for 5 second predictions.

(c) Training error for 10 second predictions.

(d) Training error for 20 second predictions.

Figure 14: Mean squared error during the training process for the three architectures predicting vibration in 1, 5, 10, and 20 seconds in the future.

Table 3: Previous Training Results

	Prediction Error (MSE)			
	1 seconds	5 seconds	10 seconds	20 seconds
Architecture I	0.000154	0.000398	0.000972	0.001843
Architecture II	0.001239	0.001516	0.001962	0.002870
Architecture III	0.000133	0.000409	0.000979	0.001717

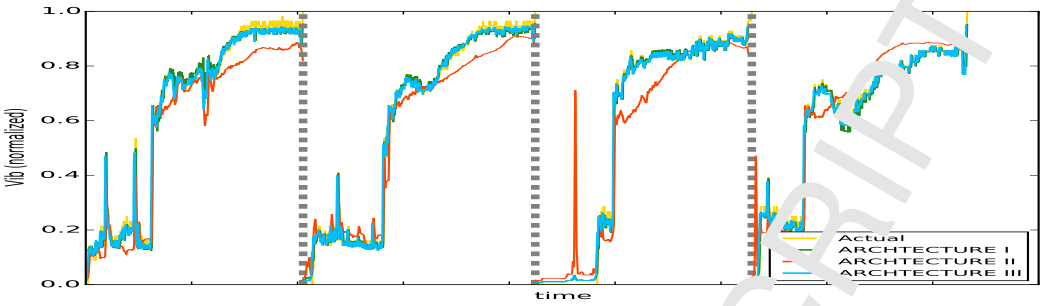
6.2. Previous Architecture Results

For Architectures I, II and III, the MSE for predicting 1 sec, 5 sec, 10 sec and, 20 sec during the training process is shown in Figure 14. Results are shown in logarithmic scale. Initially, all three architectures were trained for 575 epochs, however additional epochs were later examined for Architecture III. In these preliminary results the vibration data set was split into 28 flights in the training set, with a total of 41,431 seconds of data, and 7 flights in the testing set, with a total of 38,126 seconds of data.

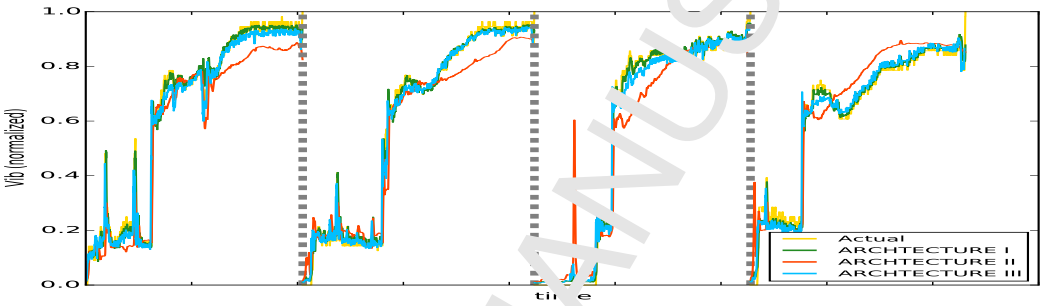
The final mean squared error on the training data is shown in Table 3. Figure 15 presents the predictions for a selection of flights from the test set, and Figures 16 provides an uncompressed example of predicting vibration 1, 5, 10 and 20 seconds in the future over a single flight from the testing data. In the multiple flight plots, each flight ends when the vibration reaches the max critical value (normalized to 1) and after which the next flight in the test set begins.

6.2.1. Results of Architecture I

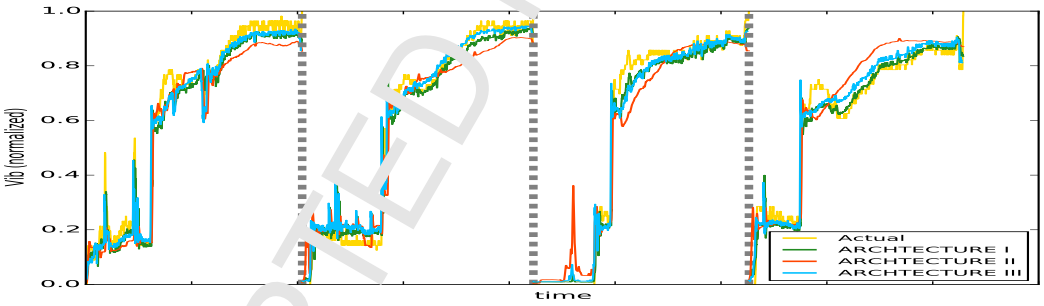
The results of this architecture, shown in Table 4, came out to be the most accurate in predicting the vibration parameter. There is more misalignment between the actual and calculated vibration values as predictions are made further in the future, as shown in Figures 15, and 16, which present predictions over a selection of test flights and over a single flight in higher resolution, respectively. This misalignment is to be



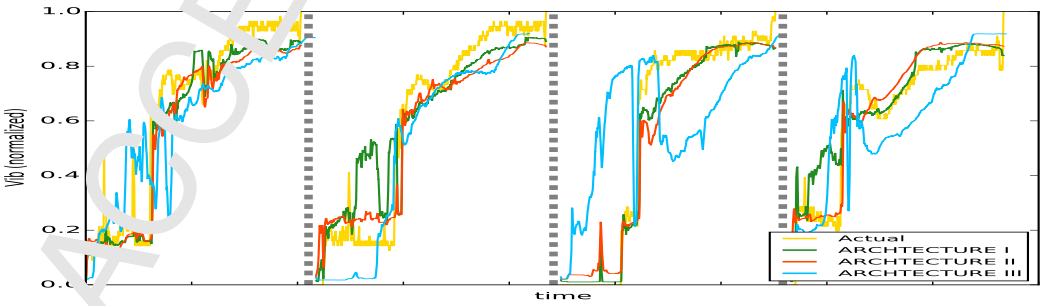
(a) Predictions for 1 second in the future.



(b) Predictions for 5 seconds in the future.

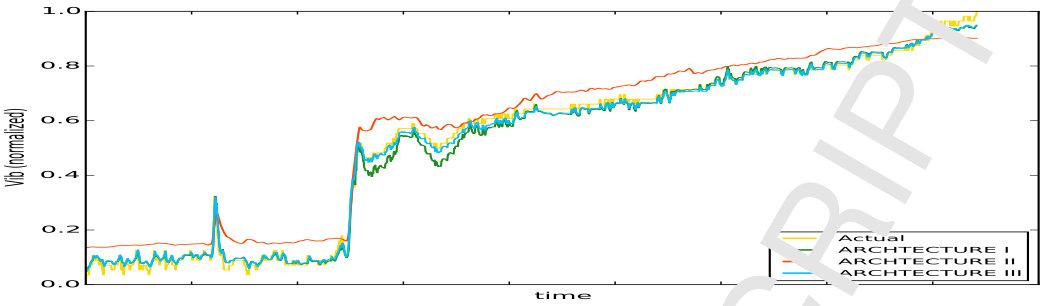


(c) Predictions for 10 seconds in the future.

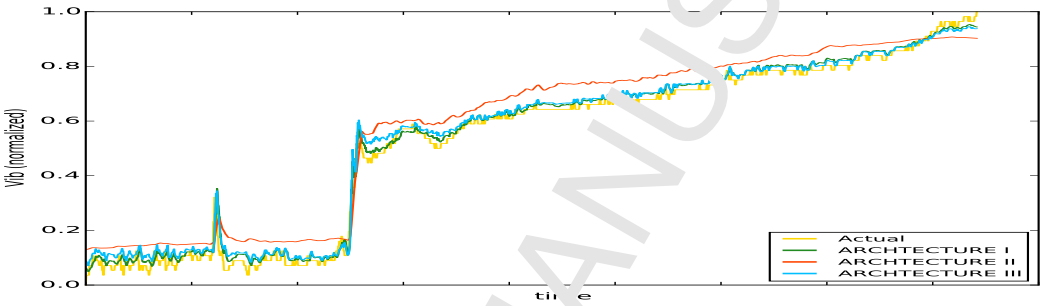


(d) Predictions for 20 seconds in the future.

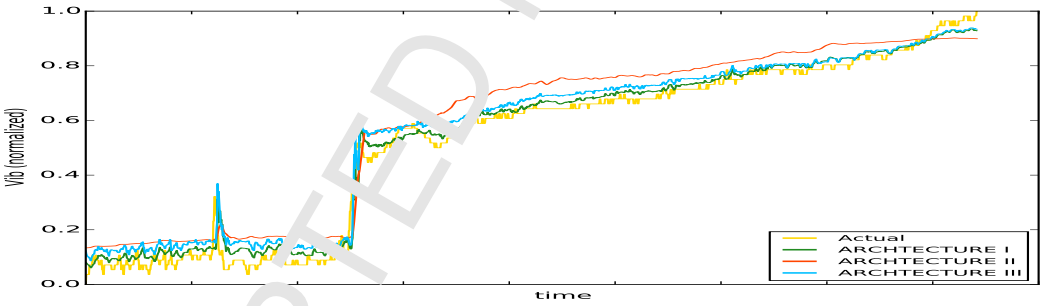
Figure 15: Plotted results for Architectures I, II, and III for 1, 5, 10 and 20 seconds in the future for a selection of test flights.



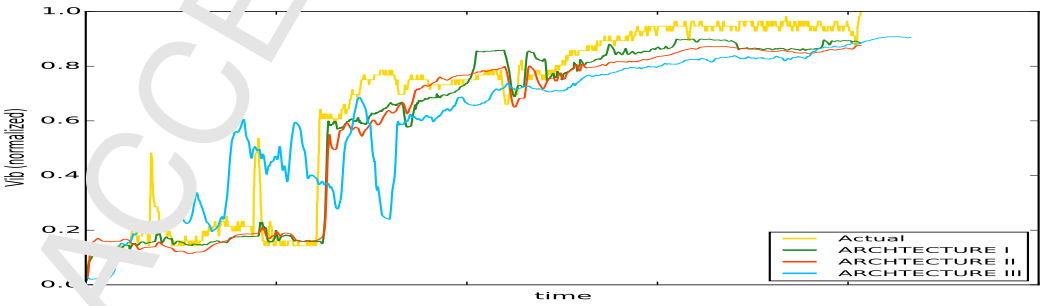
(a) Predictions for 1 second in the future.



(b) Predictions for 5 seconds in the future.



(c) Predictions for 10 seconds in the future.



(d) Predictions for 20 seconds in the future.

Figure 16: Plotted results for Architectures I, II, and III for 1, 5, 10 and 20 seconds in the future for a single test flight.

Table 4: Previous Testing Mean Squared Error

	Prediction Error (MSE)			
	1 seconds	**5 seconds**	**10 seconds**	**20 seconds**
Architecture I	0.000792	0.001165	0.002926	0.010427
Architecture II	0.010311	0.009708	0.009056	0.012560
Architecture III	0.000838	0.002386	0.004780	0.041417

Table 5: Previous Testing Mean Absolute Error

	Prediction Error (MAE)			
	1 seconds	**5 seconds**	**10 seconds**	**20 seconds**
Architecture I	0.028407	0.033048	0.055124	0.101991
Architecture II	0.098357	0.097588	0.096054	0.112290
Architecture III	0.027621	0.048056	0.070360	0.202609

expected as it is more challenging to predict further in the future. Also, it can be seen that the prediction of higher peaks is more accurate than the prediction of lower peaks, as if the neural network is tending to learn more about the max critical vibration value, which is favorable for this project.

6.2.2. Results of Architecture II

The results of this architecture, shown in Table 4, came out to be the least accurate in vibration prediction. While it managed to predict much of the vibration, its performance was weak at the peaks (either low or high) compared to the other architectures, as shown in Figures 15, and 16. However, it is also worth mentioning that somehow the lower peaks were better at some positions on the curve of this architecture, compared to the other architectures. A potential reason for the poor performance of this architecture is due to using the average of values from the LSTM second layer output, the other two architectures can weight the values from the LSTM second layer output for more accuracy.

6.2.3. Results of Architecture III

This LSTM RNN was one layer deeper and also had 20 seconds memory from the past which was not available for the other two LSTM RNNs used. Although it was the most computationally expensive and thus had the most chance for deeper learning, the results of this architecture were not as good as expected, as shown in Figures 15, and 16. The results of this architecture in Table 4 show that the prediction accuracy for this architecture was less than the more simple Architecture I.

The overall error in Table 4 for the prediction at 20 future seconds was relatively high. At some locations in the plotted curve for this architecture predicting vibrations at 20 seconds in the future, it can be seen that the calculated curve got very much higher than the actual vibration curve. This strange behavior is unique as it can be seen that the calculated vibration would rarely exceed the actual vibration for all the curves plotted for all the architectures at all scenarios, and it would be for relatively small value if occurred.

The performance of this architecture (the mean absolute error) was slightly better than the other architectures when predicting for 1 second in the future, however it performed worse in the other time scales. While a more complex architecture should have more potential to "learn" and perform better predictions, it appears that the challenge of training this negated most of these benefits. This is reinforced by Figure 14, which shows that the error of this architecture did not decrease smoothly while trained to predict for 20 seconds in the future. To investigate if this network could potentially gain further improvement if trained for more epochs, it was retrained for 1150 instead of 575 epochs. However, this failed to result in any significant improvement in predictive ability. This could potentially be a result of other issues in the training process due to a more complex search space.

6.3. Ant Colony Optimization Results

As in the prior work, Architecture I gave the most promising results, it was chosen as the initial candidate for the ACO. The ACO code was run for 1000 iterations using 200 ants. The networks were allowed to train for 575 epochs to learn and for the error curve to flatten. The minimum value for the pheromones were 1 and

the maximum was 20. The population size was equal to number number of iterations in the ACO process, *i.e.*, the population size was also 1000. Each run took approximately 4 days.

For a more rigorous examination, the 57 flights in the vibration data set were divided into 10 subsamples. The subsamples were used to cross validate the results by examining combinations utilizing nine of the subsamples as the training data set and the tenth as the testing set.

These subsamples were used to train the NOE, NARX, NBH, Architecture I and the ACO Architecture I. Figures 22 shows predictions for the different models over a selection of test flights, and Figure 23 shows predictions of a single uncompressed (higher resolution) test flight. Table 7 compares these models to Architecture I (LSTM) and the ACO optimized Architecture I (ACO). Figure 21 shows box-plot for the results shown in Table 7.

6.3.1. NOE, NARX, and NBJ Results

Somewhat expectedly, the NOE model performed the worst with with a mean error of 15.73% ($\sigma = 0.0941$). The NBJ model performed better than the NOE model with a mean error of 15.05% ($\sigma = 0.1338$), however the NARX model better than the previous two models with a mean error of 12.06 % ($\sigma = 0.0663$). This is interesting in that the NBJ model had access to actual future vibration values, unlike NOE, NARX and the LSTM models; and could be expected to perform better utilizing this information. The discrepancy in performance is likely due to the high nonlinearity in the input and target parameters, along with the difficulty of training RNNs on long time series data.

6.3.2. Architecture I Revisited

The Architecture I RNN was trained utilizing the ten subsamples to validate the results. The mean error for each of the ten subsamples (using the other two as training data) was 5.61% ($\sigma = 0.0245$).

6.3.3. Networks Regularization

Regularization [58] was implemented on the investigated LSTM, NOE, NARX, and NBJ networks to validate ACO. The used connection-dropout percent is 30%. This percent was chosen because the number connection which were subject to regularization is not very large. The results were as follows: *a) LSTM Regularization:* the obtained mean error is 7.32% ($\sigma = 0.0183$), *b) NOE Regularization:* the obtained mean error is 8.70% ($\sigma = 0.0021$), *c) NARX Regularization:* the obtained mean error is 9.40% ($\sigma = 0.0013$), and *d) NBJ Regularization:* the obtained mean error is 9.43% ($\sigma = 0.0020$).

6.3.4. Ant Colony Optimized Architecture I

When ACO optimization was used to find the optimal connections to use in the Architecture I RNN, the best version of Architecture I evolved with ACO showed an improvement of 1.34% for predictions 10 seconds in the future, reducing prediction error from 5.61% to 4.27% compared to the architecture's performance before the ACO. Figure 17 provides an example of the improvement in predictions on a single test flight, before and after the ACO optimization.

The topology of the best networks' cells are shown in Figures 18 and 19. The ACO generated mesh (as defined in Algorithm 1) used to generate this topology is shown in the matrices in Equations 16 and 17. It is worth stressing that this topology is not the complete LSTM RNN used in the utilized Architecture I, but rather applies to the individual gates in each cell. Equation 16 is used for any fully connected process and Equation 17 is used for any data-reduction process (discussed in detail in Section 4). Figures 18a, 18b, and 18c show the ACO optimized *mesh_1*, which were used within the "M1" LSTM cells (see Figure 4) in the top ten evolved LSTM RNNs. Figure 20 provides an example of how the M1 cells are updated with these connections. While the connections reduction did not show any inputs being fully eliminated, which was sought as one of the goals of the study, a significant number of connections were removed.

The evolved networks retained all the elements of *mesh_2* , represented by Equation 17, for use in the "M2" LSTM cells (see Figure 5). Figure 19 is used to show this part of the evolved mesh. For clarity, Figure 20b shows the differences between the M1 cells before and after ACO optimization. Figure 20a is simply a LSTM cell "M1" that have its gates' meshes (shown in Figure 20b, Up) substituted with the ACO meshes (shown in Figure 20b, Down). "M2" did not change from its original topology as shown in Figure 5 since all the elements in *mesh_2* after the optimization remained ones (Equation 17).

Table 6: ACO Top Thirty Evolved Networks

No.	Fitness	Number of M1 Connections	Number of M2 Connections	Total Number of Connections	No.	Fitness	Number of M1 Connections	Number of M2 Connections	Total Number of Connections
1	0.034888	137	16	11650	16	0.036795	140	16	11890
2	0.034917	136	16	11570	17	0.036820	145	16	12290
3	0.035851	141	16	11970	18	0.036901	140	16	11890
4	0.036063	146	16	12370	19	0.036932	131	16	11170
5	0.036067	143	16	12130	20	0.036953	142	16	12050
6	0.036337	140	16	11890	21	0.037001	141	16	11970
7	0.036535	136	16	11570	22	0.037040	145	16	12290
8	0.036582	140	16	11890	23	0.037041	147	16	12450
9	0.036588	133	16	11330	24	0.037082	133	16	11330
10	0.036647	134	16	11410	25	0.037106	142	16	12050
11	0.036715	135	16	11490	26	0.037114	135	16	11490
12	0.036727	143	16	12130	27	0.037134	137	16	11650
13	0.036730	147	16	12450	28	0.037142	138	16	11730
14	0.036787	143	16	12130	29	0.037145	144	16	12210
15	0.036788	137	16	11650	30	0.03716	13	16	11810

The colored nodes in Figure 18a are the input nodes (first line of nodes) at the Main, Input, Forget, and Output gates at the "M1" cells (Figure 4). The diamond nodes in Figure 18a are the hidden layer nodes (second line of nodes) at the Main, Input, Forget, and Output gates at the "M1" cells (Figure 4). The diamond nodes are also the input nodes at the Main, Input, Forget, and Output gates at the"M2" cells (Figure 5). The last single node in Figure 19 is the output of the gates in the "M2" cells.

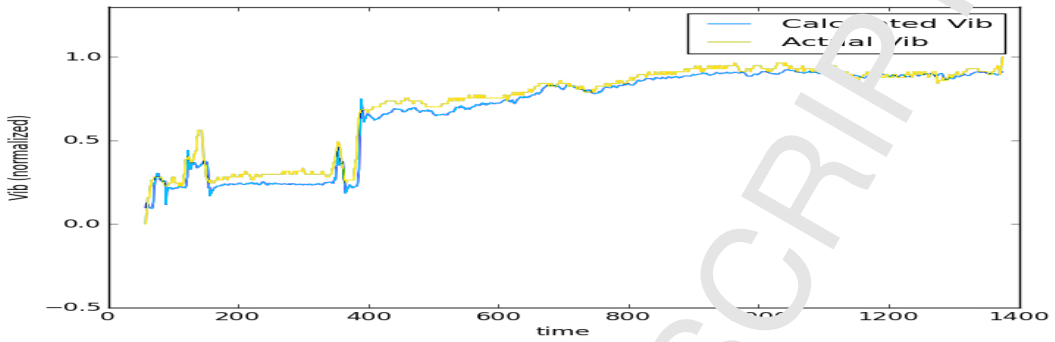
Returning to an initial question of how the number of the connections in the network affects the soundness of the results, Table 6 shows the top thirty evolved networks with respect to the fitnesses they provide. The table also shows the total number of connections in both *mesh_1* (first set of connections in the generated mesh) and *mesh_2* (second set of connections in the generated mesh), and the total number weights (connections) in the networks. Comparing these values to the total number of weights in a fully connected Architecture I type network, as shown in Table 2, it is found that total number of weights were reduced by 42% to 45% in the top 30 networks.

$$mesh\_1 = \begin{array}{c|cccccccccccccccc} & h1 & h2 & h3 & h4 & h5 & h6 & h7 & h8 & h9 & h10 & h11 & h12 & h13 & h14 & h15 & h16 \\ i1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ i2 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ i3 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ i4 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ i5 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ i6 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ i7 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ i8 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ i9 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ i10 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ i11 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ i12 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ i13 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ i14 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ i15 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ bias & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{array} \tag{16}$$

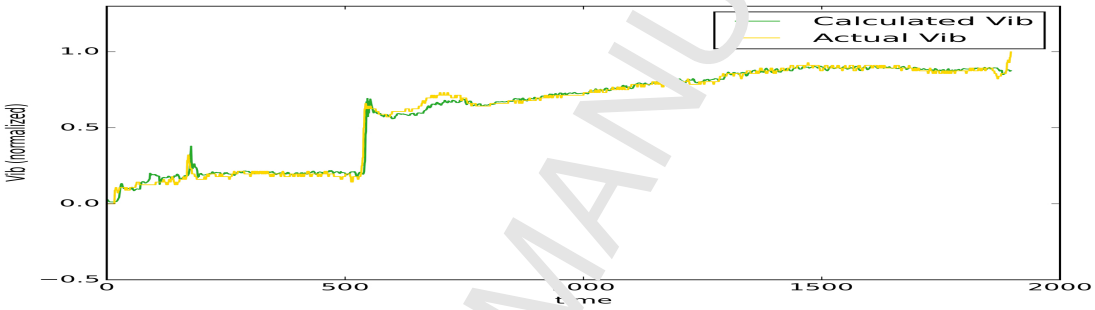
$$mesh\_2 = \begin{array}{c|cccccccccccccccc} & h1 & h2 & h3 & h4 & h5 & h6 & h7 & h8 & h9 & h10 & h11 & h12 & h13 & h14 & h15 & h16 \\ & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array} \tag{17}$$

7. Discussion and Future Work

The results have shown that the ACO approach for optimizing the gates within LSTM cells can dramatically reduce the number of connections required, while at the same time improve the predictive ability of the recurrent neural network. However, as much of the matrix in Equation 16 is sparse, none of the rows of this matrix had all their elements equal to zero, meaning that none of the inputs ended up being fully removed (which was a potential means for improving predictions, as discussed in Subsection 3.1.2). This is a indication that all the chosen parameters actually had a positive contributing influence on the vibration. On the other hand, it suggests that having extraneous connections can increase the difficulty of appropriately training the LSTM RNN, resulting in less predictive ability (as in the case of the original unoptimized LSTM RNN architectures). In addition, the results showed that ACO outperformed the connection-dropout technique used in neural network regularization.



(a) Unoptimized: single test flights

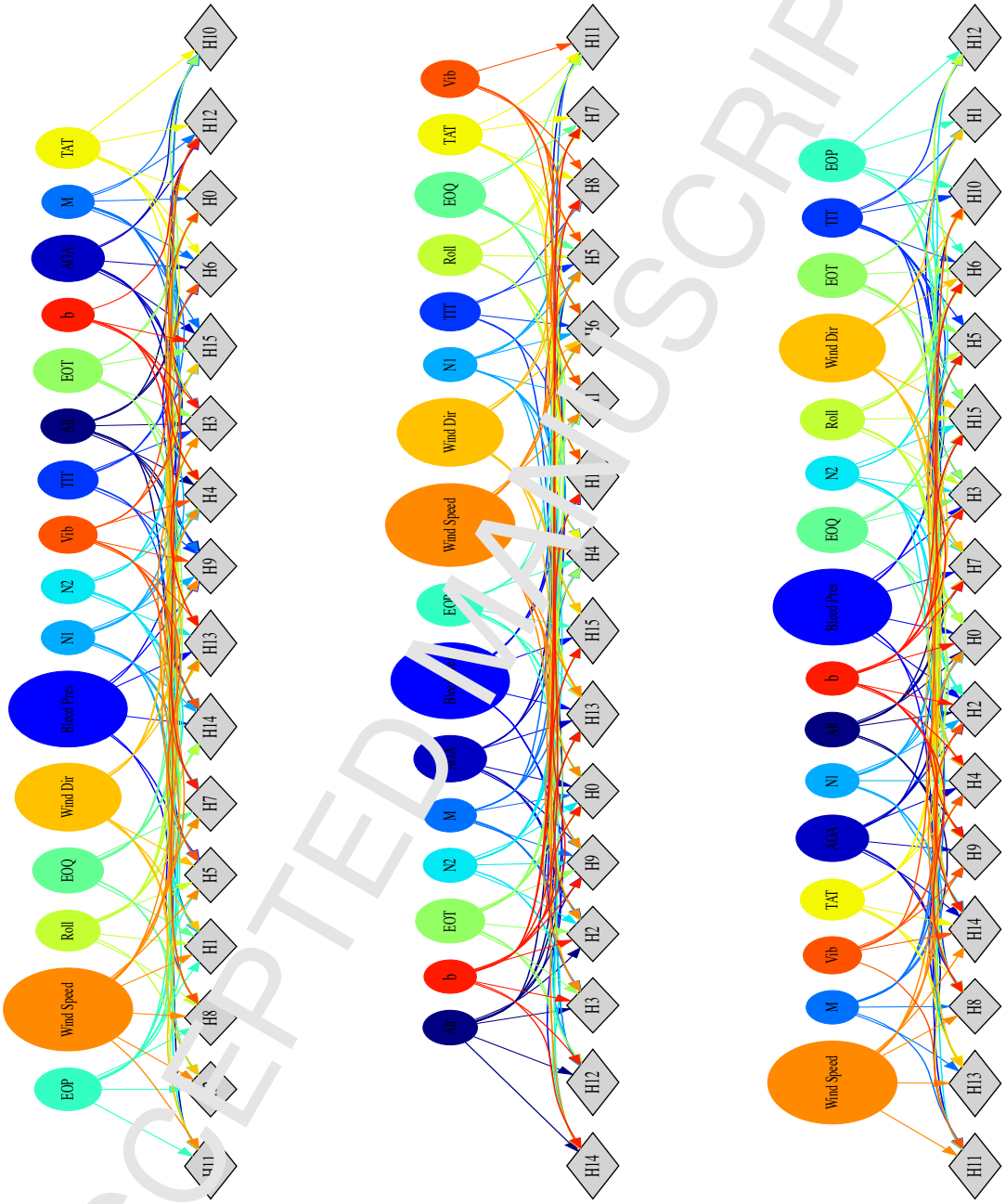


(b) Optimized: single test flights

Figure 17: Plotted results for predicting ten seconds in the future.

Table 7: 10-Fold Cross Validation Results

	Prediction Errors (MAE)								
	LSTM	**LSTM_Reg**	**NOE**	**NARX**	**NBJ**	**LSTM_ACO**	**NOE_Reg**	**NARX_Reg**	**NBJ_Reg**
Fold 1	8.98%	5.68%	12.45%	10.1%	9.28%	8.11%	9.14%	8.63%	9.24%
Fold 2	3.68%	5.04%	16.37%	16.44%	15.88%	3.39%	9.63%	8.58%	9.52%
Fold 3	7.28%	6.35%	14.19%	8.07%	7.81%	5.65%	9.25%	8.30%	9.41%
Fold 4	4.71%	7.95%	8.56%	5.18%	6.21%	4.15%	9.42%	8.70%	9.38%
Fold 5	10.76%	6.89%	15.13%	15.67%	20.10%	5.87%	9.56%	8.58%	9.41%
Fold 6	5.93%	9.40%	13.36%	7.56%	7.92%	4.05%	9.78%	8.73%	9.47%
Fold 7	3.4%	9.34%	18.03%	10.44%	13.02%	2.69%	9.60%	8.65%	9.19%
Fold 8	3.58%	5.29%	42.14%	29.27%	53.16%	3.03%	9.32%	9.01%	9.53%
Fold 9	4.1%	11.11%	10.47%	7.92%	10.57%	3.35%	9.40%	8.75%	9.60%
Fold 10	4.62%	8.33%	6.61%	9.97%	6.55%	2.43%	9.21%	9.06%	9.28%
Mean	**0.0561**	**0.0762**	**0.1573**	**0.1206**	**0.1505**	**0.0427**	**0.0870**	**0.0940**	**0.0943**
Std. Dev.	**0.0245**	**0.0183**	**0.0941**	**0.0663**	**0.1338**	**0.0168**	**0.0021**	**0.0013**	**0.0020**



(a) ACO Architecture I First Best Fitness Mesh: 155 connections.

(b) ACO Architecture I Second Best Fitness Mesh: 152 connections.

(c) ACO Architecture I Third Best Fitness Mesh: 160 connections.

Figure 18: ACO Architecture I Best Fitness Topologies' Meshes (Equation 16) for at "M1" (Figure 4) LSTM cells: 1000 Iterations, and 200 Ants.

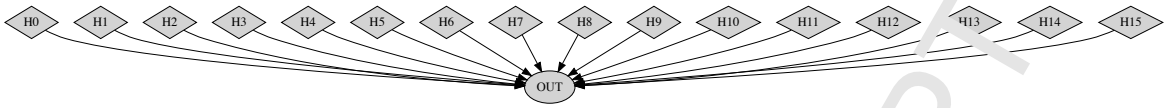


Figure 19: ACO Architecture I Best Fitness Topology's mesh (Equation 17) at "M2" (Figure 5) LSTM cells: 1000 Iterations, and 200 Ants.

This approach can open the door to further identify the highest contributors to the vibration problem by examining the number of the connections between the input neurons and the hidden layers, along with the magnitude of those weights. Furthermore, the combined effect of multiple parameters can also be investigated by looking at the input neurons connected to a certain hidden layer's neuron. This would give an idea about the effect those input neurons, which represents the input parameters have on the vibration as a final output, which could aid in discovering actual cause of the vibration event.

This also opens up the potential for significant future work. While the optimized LSTM RNNs did not remove any input connections, by increasing the number of flight parameters used as input (even using all available parameters), the algorithm has the potential to determine which parameters contribute most to the predictive ability, instead of relying on *a priori* expert knowledge to select parameters. This can be investigated by adding additional flight parameters (such as those which should not effect vibration) as inputs to the RNNS and see if the ant colony optimization eliminates them. Further, in this work one mesh of connections was generated and then used in all the LSTM cell gates at all time-steps. The future work will consider the meshes in the four gates of the LSTM cells at the various LSTM time-steps as variable and will apply the ACO on each of them simultaneously in every ACO iteration.

Future work will also consider optimizing the LSTM RNN structure. The connections of the structure shown in Figures 9 and 7 will be subject to the ACO process along with the optimization of the connections within the LSTM cells. This has the potential to make a large step forward in the evolution of the LSTM RNNs as it will allow for connections between non-adjacent cells, and potentially even dropping out certain unused cells and potentially even full layers from the LSTM RNN.

This work does also share similarities with dropout strategies for recurrent neural networks [58]. For future work, it will be worth investigating training RNNs with permanently removed connections (as done in this work) provides benefit over using a regularization method. It may also be possible to combine the two strategies, using dropout as a regularizer while training the RNNs in the ACO process.

Lastly, work investigating the tuning of the ACO hyperparameters can be done to improve how quickly the algorithm converges to optimal LSTM RNN structures. For example, modifying the number of ants, reducing pheromones on paths from LSTM RNNs with lower fitness, and periodically refreshing the pheromones levels by decreasing all of its levels by certain amount.

Acknowledgments

We very much appreciate the help, patience and support of ***Mr. Aaron Bergstrom*** of the University of North Dakota's Computational Research Center (CRC). This work used the high performance computing clusters at the CRC, where Mr. Bergstrom (the North Dakota University System HPC Specialist, UND Big Data Project Coordinator, and UND Campus Champion) offered his time and effort to much facilitate it. It was a pleasure working with him.

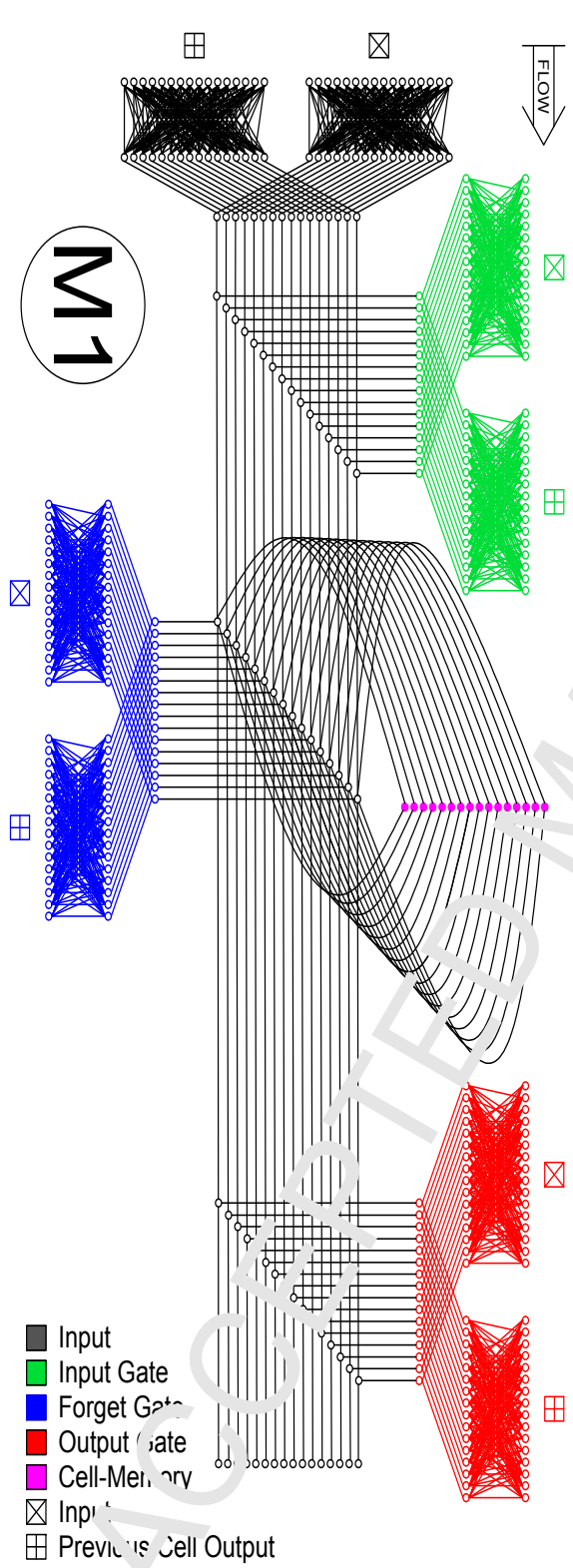
References

[1] A. V. Srinivasan, Flutter and resonant vibration characteristics of engine blades, The American Society of Mechanical Engineers.
URL http://www.energy.kth.se/compedu/webcompedu/WebHelp

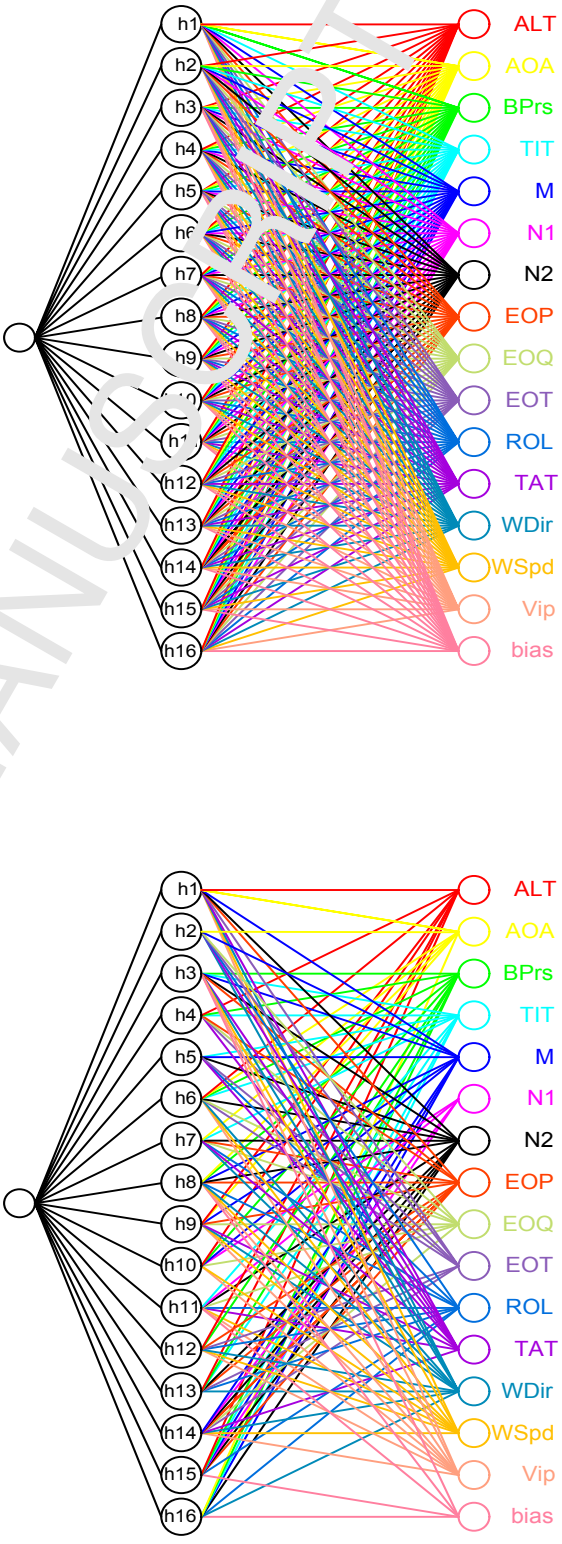
[2] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural computation 9 (8) (1997) 1735–1780.

[3] L. Di Persio, O. Honchar, Artificial neural networks approach to the forecast of stock market price movements.

[4] S. Hochrieter & J. Schmidhuber, Long Short Term Memory, Neural Computation 9(8):1735-1780.



(a) ACO Architecture I First Best Fitness Network: LSRM M1 cell.



(b) Meshes before (up) and after (down) optimization.

Figure 20: An example of the M1 cell before and after optimization.

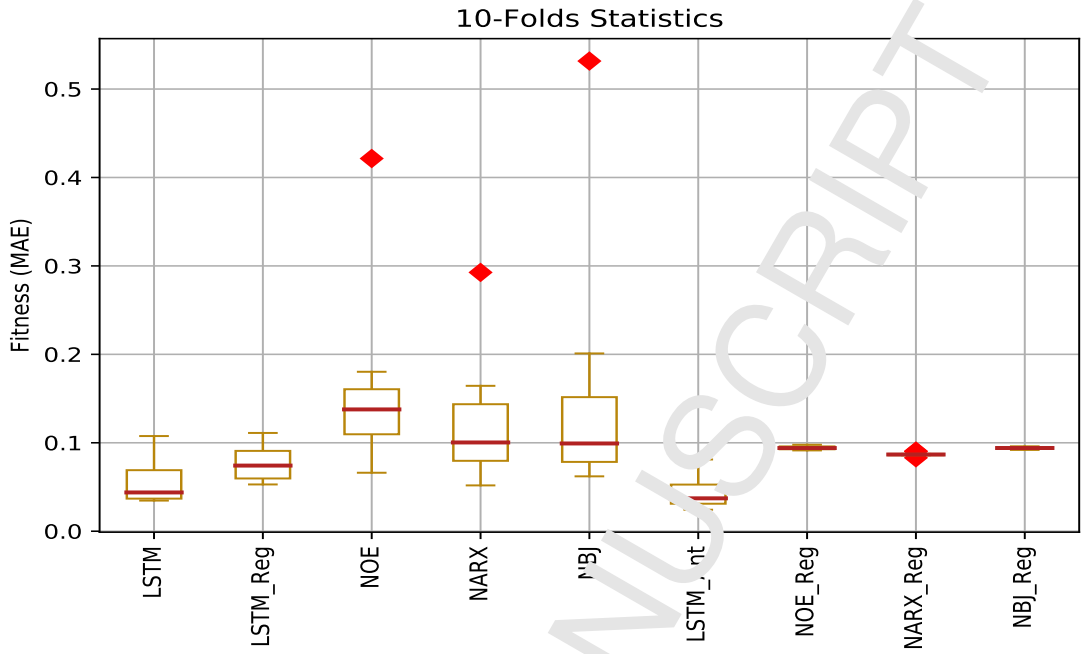


Figure 21: Box-plot: 10-Fold Cross Validation Results

[5] M. Felder, A. Kaifel, A. Graves, Wind power prediction using mixture density recurrent neural networks, in: Poster Presentation gehalten auf der European Wind Energy Conference, 2010.

[6] E. Choi, M. T. Bahadori, J. Sun, Doctor ai: Predicting clinical events via recurrent neural networks, arXiv preprint arXiv:1511.05942.

[7] N. Maknickienė, A. Maknickas, Application of neural network for forecasting of exchange rates and forex trading, in: The 7th international scientific conference" Business and Management, 2012, pp. 10–11.

[8] X. Yao, Evolving artificial neural networks, Proceedings of the IEEE 87 (9) (1999) 1423–1447.

[9] N. T. Siebel, J. Botel, G. Sommer, Efficient neural network pruning during neuro-evolution, in: Neural Networks, 2009. IJCNN 2009. International Joint Conference on, IEEE, 2009, pp. 2920–2927.

[10] J. Schmidhuber, Deep learning in neural networks: An overview, Neural networks 61 (2015) 85–117.

[11] B.-T. Zhang, H. Muhlenbein, Evolving optimal neural networks using genetic algorithms with occam's razor, Complex systems 7 (3) (1993) 199–220.

[12] N. F. Kohl, Learning in fractured problems with constructive neural network algorithms.

[13] S. Whiteson, Improving reinforcement learning function approximators via neuroevolution, in: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems, ACM, 2005, pp. 1386–1386.

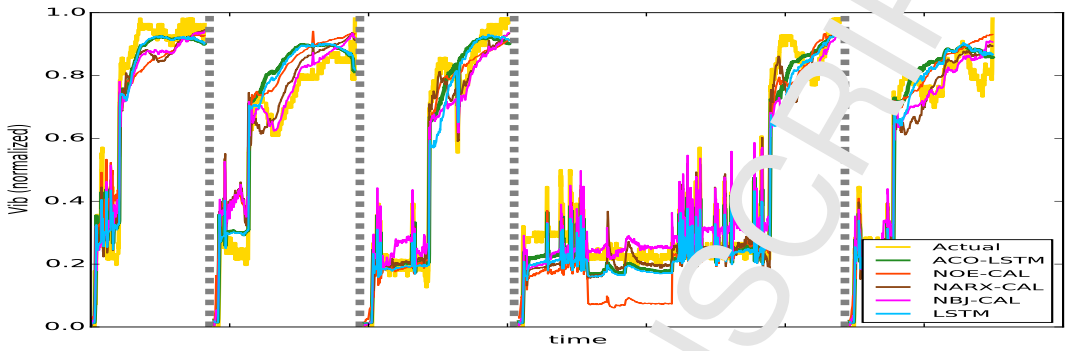
[14] D. Floreano, P. Dürr, C. Mattiussi, Neuroevolution: from architectures to learning, Evolutionary Intelligence 1 (1) (2008) 47–62.

[15] A. J. Turner, J. F. Miller, The importance of topology evolution in neuroevolution: a case study using cartesian genetic programming of artificial neural networks, in: Research and Development in Intelligent Systems XXX, Springer, 2013, pp. 213–226.

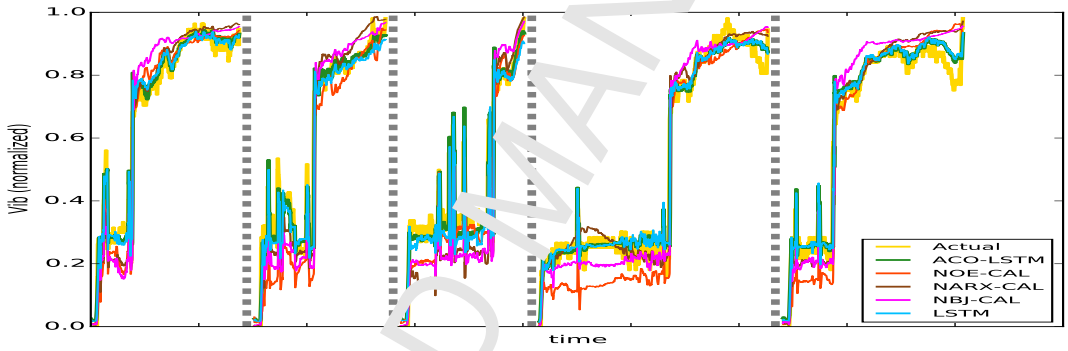
[16] T. Desell, S. Elsaghar, J. Higgins, B. Wild, Evolving deep recurrent neural networks using ant colony optimization, in: European Conference on Evolutionary Computation in Combinatorial Optimization, Springer, 2015, pp. 86–98.

[17] A. ElSaid, B. Wild, J. Higgins, T. Desell, Using lstm recurrent neural networks to predict excess vibration events in aircraft engines, in: e-Science (e-Science), 2016 IEEE 12th International Conference on, IEEE, 2016, pp. 260–269.

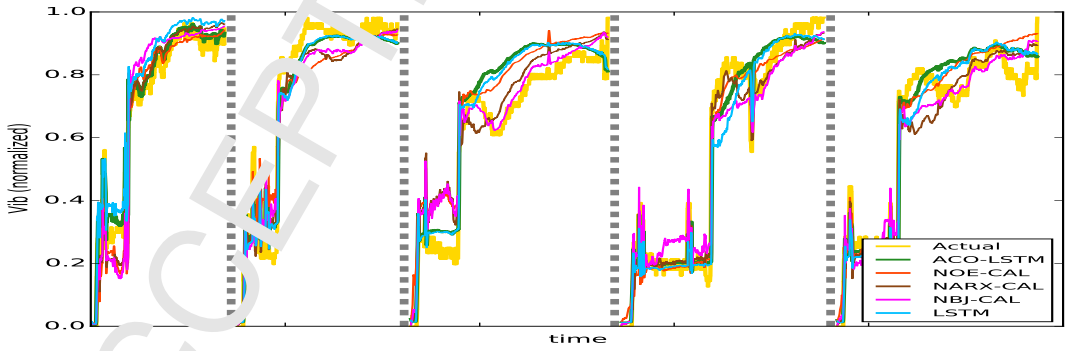
[18] M. Dorigo, V. Maniezzo, A. Colorni, Ant system: optimization by a colony of cooperating agents, IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 26 (1) (1996) 29–41.



(a) Subsample 1

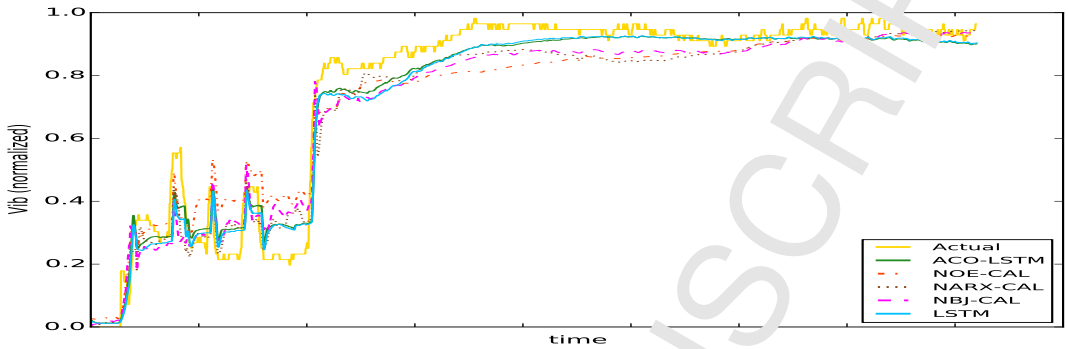


(b) Subsample 2

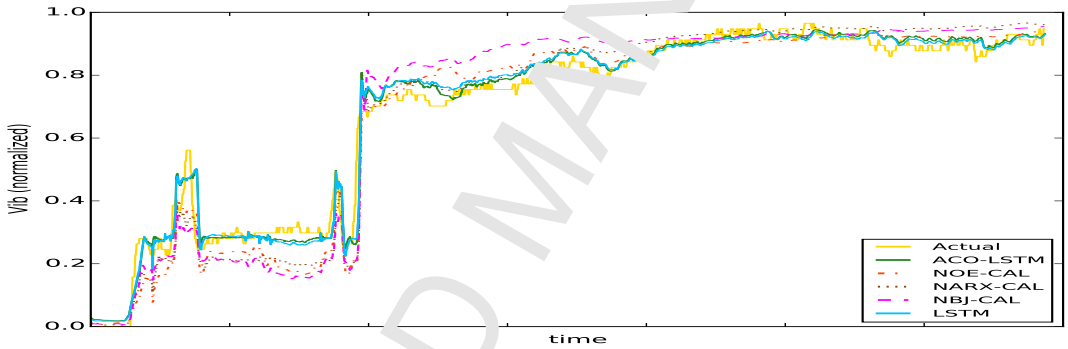


(c) Subsample 3

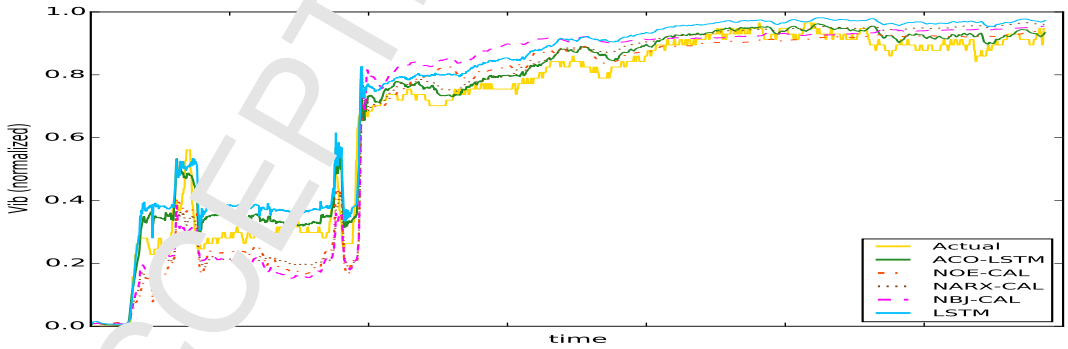
Figure 22: Results of the K-fold cross validation subsamples predicting vibration ten seconds in the future for a selection of 4 flights.



(a) Subsample 1



(b) Subsample 2



(c) Subsample 3

Figure 23: Results of the K-fold cross validation subsamples predicting vibration ten seconds in the future for an individual flight.

[19] M. Dorigo, L. M. Gambardella, Ant colony system: a cooperative learning approach to the traveling salesman problem, IEEE Transactions on evolutionary computation 1 (1) (1997) 53–66.

[20] L. Bianchi, L. M. Gambardella, M. Dorigo, An ant colony optimization approach to the probabilistic traveling salesman problem, in: International Conference on Parallel Problem Solving from Nature, Springer, 2002, pp. 883–892.

[21] M. Manfrin, M. Birattari, T. Stützle, M. Dorigo, Parallel ant colony optimization for the traveling salesman problem, in: International Workshop on Ant Colony Optimization and Swarm Intelligence, Springer, 2006, pp. 224–234.

[22] K. Socha, Aco for continuous and mixed-variable optimization. in ant colony optimization and swarm intelligence, Future Generation Computer Systems (2004) 25–36.

[23] K. Sochaand & M. Dorigo, Bottom hole pressure estimation using evolved neural networks by real coded ant colony optimization and genetic algorithm, Journal of Petroleum Science and Engineering, 77(3):375385.

[24] K. Socha, Ant colony optimisation for continuous and mixed-variable domains, VDM Publishing Saarbrücken, 2009.

[25] G. Bilchev, I. Parmee, The ant colony metaphor for searching continuous design spaces, Evolutionary Computing (1995) 25–39.

[26] N. Monmarch and G. Venturini and M. Slimane, On how pachycondyla apicalis ants suggest a new search algorithm, Future Generation Computer Systems 16 (2000) 937–946.

[27] J. Dréo, P. Siarry, A new ant colony algorithm using the heterarchical concept aimed at optimization of multiminima continuous functions, in: International Workshop on Ant Algorithms, Springer, 2002, pp. 216–221.

[28] R. Ashena and J. Moghadasi, Ant colony optimization for continuous domains, European journal of operational research, 185(3):11551173.

[29] C. Blum, K. Socha, Training feed-forward neural networks with ant colony optimization: An application to pattern classification, in: Fifth International Conference on Hybrid Intelligent Systems (HIS'05), IEEE, 2005, pp. 6–pp.

[30] J.-B. Li and Y.-K. Chung, A novel back-propagation neural network training algorithm designed by an ant colony optimization, In Transmission and Distribution Conference and Exhibition: Asia and Pacific, 2005 IEEE/PES, pages 15. IEEE.

[31] M. Unal, M. Onat, and A. Bal, Cellular neural network training by ant colony optimization algorithm, In Signal Processing and Communications Applications Conference (SIU), 2010 IEEE 18th, pages 471474. IEEE.

[32] J. Moubray, Reliability-centered maintenance, Industrial Press Inc., 1997.

[33] C. Chatfield, The analysis of time series: an introduction, CRC press, 2016.

[34] N. A. Boukary, A comparison of time series forecasting learning algorithms on the task of predicting event timing, Ph.D. thesis, Royal Military College of Canada (2015).

[35] H. Goel, I. Melnyk, N. Oza, B. Matthews, A. Banerjee, Multivariate aviation time series modeling: Vars vs. lstms.

[36] V. M. Ranković, I. Ž. Nikolić, Identification of nonlinear models with feed forward neural network and digital recurrent network, FME Transactions 36 ( ) (2008) 87–92.

[37] A. Nairac, N. Townsend, R. Carr, S. King, P. Cowley, L. Tarassenko, A system for the analysis of jet engine vibration data, Integrated Computer-Aided Engineering 6 (1) (1999) 53–66.

[38] D. A. Clifton, P. R. Bannister, L. Tarassenko, A framework for novelty detection in jet engine vibration data, in: Key engineering materials, Vol. 347, Trans Tech Publ, 2007, pp. 305–310.

[39] F. A. Gers, N. N. Schraudolph, J. Schmidhuber, Learning precise timing with lstm recurrent networks, Journal of machine learning research 3 (Aug) (2002) 115–143.

[40] T. Desell, S. Clachar, J. Higgins, B. Wild, Evolving neural network weights for time-series prediction of general aviation flight data, in: T. Bartz-Beielstein, J. Branke, B. Filipič, J. Smith (Eds.), Parallel Problem Solving from Nature - PPSN XIII, Vol. 8672 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 771–781. doi:10.1007/978-3-319-10762-2_76.
URL http://dx.doi.org/10.1007/978-3-319-10762-2_76

[41] J. Kennedy, Particle swarm optimization, in: Encyclopedia of machine learning, Springer, 2011, pp. 760–766.

[42] R. Poli, J. Kennedy, T. Blackwell, Particle swarm optimization, Swarm intelligence 1 (1) (2007) 33–57.

[43] R. Storn, K. Price, Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces, Journal of global optimization 11 (4) (1997) 341–359.

[44] C. Blum, X. Li, Swarm intelligence in optimization, in: Swarm Intelligence, Springer, 2008, pp. 43–85.

[45] M. Dorigo, M. Birattari, T. Stutzle, Ant colony optimization, IEEE computational intelligence magazine 1 (4) (2006) 28–39.

[46] M. Dorigo, T. Stützle, Ant colony optimization: overview and recent advances, in: Handbook of metaheuristics, Springer, 2010, pp. 227–263.

[47] M. Dorigo and L. M. Gambardella, Ant colonies for the travelling sales man problem, BioSystems, 43(2):7381.

[48] Felix A. Gers, Jrgen Schmidhuber, and Fred Cummins, Learning to Forget: Continual Prediction with LSTM, Neural Computation, Vol. 12, No. 10 , Pages 2451-2471.

[49] F. A. Gers, D. Eck, J. Schmidhuber, Applying lstm to time series predictable through time-window approaches, in: Neural Nets WIRN Vietri-01, Springer, 2002, pp. 193–200.

[50] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, T. Darrell, Long-term recurrent convolutional networks for visual recognition and description, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2015, pp. 2625–2634.

[51] L. Chao, J. Tao, M. Yang, Y. Li, Z. Wen, Audio visual emotion recognition with temporal alignment and perception attention, arXiv preprint arXiv:1603.08321.

[52] D. Eck, J. Schmidhuber, A first look at music composition using lstm recurrent neural networks, Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale 103.

[53] K. O. Stanley, R. Miikkulainen, Evolving neural networks through augmenting topologies, Evolutionary computation 10 (2) (2002) 99–127.

[54] M. Annunziato, M. Lucchetti, S. Pizzuti, Adaptive systems and evolutionary neural networks: a survey, Proc. EUNITE02, Albufeira, Portugal.

[55] H. Larochelle, Y. Bengio, J. Louradour, P. Lamblin, Exploring strategies for training deep neural networks, Journal of Machine Learning Research 10 (Jan) (2009) 1–40.

[56] E. R. Kandel, J. H. Schwartz, T. M. Jessell, S. Siegelbaum, A. J. Hudspeth, Principles of neural science, Vol. 4, McGraw-hill New York, 2000.

[57] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting., Journal of machine learning research 15 (1) (2014) 1929–1958.

[58] W. Zaremba, I. Sutskever, O. Vinyals, Recurrent neural network regularization, arXiv preprint arXiv:1409.2329.

[59] J. P. Lewis, Fast normalized cross-correlation, in: Vision interface, 1995, pp. 120–123.

[60] L. Dalcín, R. Paz, M. Storti, J. D'Elía, Mpi for python: Performance improvements and mpi-2 extensions, Journal of Parallel and Distributed Computing 68 (5) (2008) 655–662.

[61] B. Szymanski, T. Desell, C. Varela, The effect of heterogeneity on asynchronous panmictic genetic search, in: Proc. of the Seventh International Conference on Parallel Processing and Applied Mathematics (PPAM'2007), LNCS, Gdansk, Poland, 2007.

[62] T. Desell, D. Anderson, M. Magdon-Ismail, B. S. Heidi Newberg, C. Varela, An analysis of massively distributed evolutionary algorithms, in: The 2010 IEEE congress on evolutionary computation (IEEE CEC 2010), Barcelona, Spain, 2010.

[63] Theano Development Team, Theano: A Python framework for fast computation of mathematical expressions, arXiv e-prints abs/1605.02688.
URL http://arxiv.org/abs/1605.02688

[64] R. Pascanu, T. Mikolov, Y. Bengio, On the difficulty of training recurrent neural networks, in: International Conference on Machine Learning, 2013, pp. 1310–1318.

[65] P. J. Werbos, Backpropagation through time: what it does and how to do it, Proceedings of the IEEE 78 (10) (1990) 1550–1560.

[66] O. Nelles, Nonlinear system identification: from classical approaches to neural networks and fuzzy models, Springer Science & Business Media, 2013.

*Highlights (for review)

Highlights:

- Long Short Term Memory RNN to predict excessive turbine engine vibration events.
- Ant Colony Optimization to optimize the best previously used LSTM RNN fixed topology.
- Evolved LSTM reduced engine vibration mean prediction error from 6.38% to 5.01%.
- ACO dramatically reduced the number of weights from 21,170 to 11,650.