Using Ant Colony Optimization to Optimize Long Short-Term Memory Recurrent Neural Networks

AbdElRahman ElSaid*† abdelrahman.elsaid@und.edu
Fatima El Jamiy*† fatima.eljamiy@und.edu
James Higgins‡† jhiggins@aero.und.edu
Brandon Wild‡† bwild@aero.und.edu
Travis Desell*† tdesell@cs.und.edu

ABSTRACT

This work examines the use of ant colony optimization (ACO) to improve long short-term memory (LSTM) recurrent neural networks (RNNs) by refining their cellular structure. 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, which evolved 1000 different LSTM cell structures using 208 cores over 5 days. The new evolved LSTM cells showed an improvement in prediction accuracy of 1.37%, reducing the mean prediction error from 6.38% to 5.01% when predicting excessive engine vibrations 10 seconds in the future, while at the same time dramatically reducing the number of trainable weights from 21,170 to 11,650. The ACO 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 11.45%, 8.47% and 9.77%, respectively. The ACO algorithm employed could be utilized to optimize LSTM RNNs for any time series data prediction task.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; **Genetic algorithms**;

KEYWORDS

Neuroevolution, Recurrent Neural Networks, Long Short-Term Memory, Ant Colony Optimization, Time Series Data Prediction

ACM Reference Format:
AbdElRahman ElSaid, Fatima El Jamiy, James Higgins, Brandon Wild, and Travis Desell. 2018. Using Ant Colony Optimization to Optimize Long Short-Term Memory Recurrent Neural Networks. In *GECCO '18: Genetic and Evolutionary Computation Conference, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205455.3205637

*Department of Computer Science
†University of North Dakota
‡Department of Aviation

GECCO '18, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5618-3/18/07.
https://doi.org/10.1145/3205455.3205637

1 INTRODUCTION

This work proposes an algorithm based on ant colony optimization (ACO) which can optimize the cell structure of long short-term memory (LSTM) recurrent neural networks. LSTM RNNs have been used successfully in many studies on involving time series data [2, 6, 8, 14, 19] and were chosen by this study to examine them as a solution to predicting aircraft engine vibration. Aircraft engine vibration provides both a challenging learning task as well as a strong motivation – if accurate predictions can be made far enough in the future, it is possible to develop warning systems with the potential to save time, effort, money, and human lives in the aviation industry.

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.

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 pose 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 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 [10], *"Learning to store information over extended period of time intervals via recurrent backpropagation takes a very long time, mostly due to insufficient, decaying error back flow."* While this drawback hindered 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.

Even so, large LSTM RNNs can be challenging to train and design. The structure of the LSTM RNN and how well they can be trained are highly correlated, yet not in an readily apparent way. Neuroevolution algorithms provide a way to overcome these challenges by automating the design and optimization process of a neural network. In traditional neuroevolution [28], 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 [9, 12, 26], as weights are not the only key parameter for best performance of neural networks [24]. 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 [5] based algorithm, which allows for keeping the superstructure of the LSTM RNN fixed while refining its individual components.

Using k-fold cross validation (k = 3), results demonstrate that the evolved LSTM architecture increase the performance by 1.37% over the non-optimized architecture 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 the number of trainable connections (the number of weights was reduced from 21,170 to 11,650). These results also significantly outperform traditional methods, such as Nonlinear Output Error (NOE), Nonlinear AutoRegression with eXogenous (NARX) input, and 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 Evolutionary Optimization Methods

Several methods for evolving topologies along with weights have been searched and deployed. In [21], 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 [1, 11, 13]. 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 [5]. More recent works by Miikkulainen, Rawal *et al.*have extended NEAT to evolve LSTM RNNs with success on a sequence recall task [18] and CoDeepNeat to utilize LSTM variants for image captioning [15]. To our knowledge, LSTM RNNs have not been evolved for time series data prediction, as done in this work.

2.2 RNN Regularization vs. ACO Optimization

Srivastava *et al.* have demonstrated the ability to utilize Dropout, a popular method for regularization of convolutional neural networks [20], to be applied as a regularizer for recurrent neural networks [27]. 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 algorith, 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 [27], 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

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. A subset of the FDR parameters were chosen based on the likelihood of their contribution to the vibration based on aerodynamics/turbo-machinery expert knowledge. These paremeters were validated with a simple fully connected one layer feed forward neural network, which provided results 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:

(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]

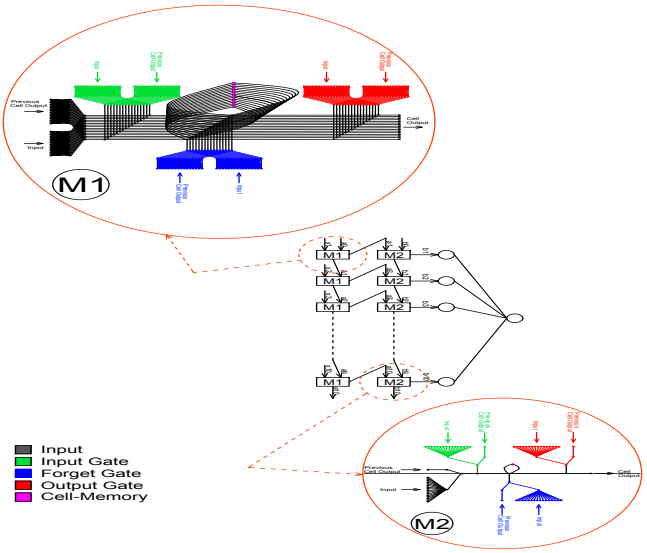


Figure 1: The base LSTM RNN archiecture used in this work.

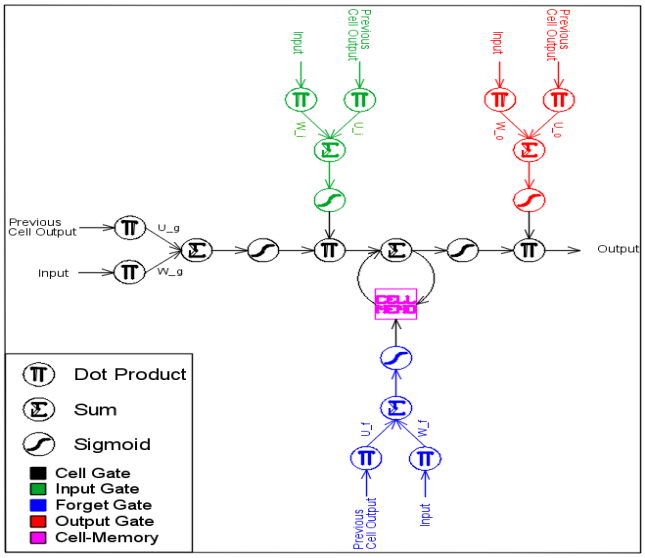


Figure 2: LSTM cell design

3.2 Recurrent Neural Network Design

This work utilizes as base a LSTM RNN architecture (Figure 1) which has been shown by ElSaid *et al.* to have strong predictive accuracy on engine vibration data in a comparison of various LSTM RNN architectures [7]. The first level of the architecture uses the 15 selected parameters from ten time series (the current time instant and the past nine) as input. 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 is 10 nodes, a node from each cell. These nodes are fed to a final neuron in the third level to compute the prediction of the whole network.

All the utilized architectures follow the common LSTM cell design (see Figure 2). Cells consist of the following gates: *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.

There are two variations of this common design used which are labeled the *'M1'* and *'M2'* cells. 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. M1 cells have a fully connected (or *m*esh) layer for each of the LSTM gates, 15 inputs + 1 bias fully connected to 15 outputs; while the M2 cells have a data reduction, 15 inputs + 1 bias to 1 output. In total, this architecture has 21,170 trainable weights.

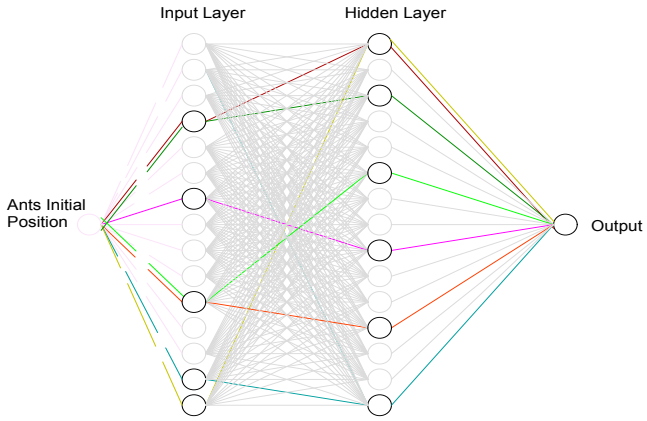


Figure 3: Schematic of Neural Network Structure after AOC. Different ants can select the same edge through the network, however that will not result in duplicate edges in the generated LSTM RNN.

4 EVOLVING LSTM RNN CELLS USING ANT COLONY OPTIMIZATION

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 determines 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 edges are removed and a design network is generated based on the new cell topology. Figure 3 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 ACO generated topology is used for each of these 8

input gates. Figure 10 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 M1 cells in black, green, blue, and red 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 M2 cells in black, green, blue, and red, 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. 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 [3] 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 [4, 22]. 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 from whatever the pheromone values currently present are.

In detail, the algorithm beings with workers requesting LSTM RNNs from an uninitialized population. In this case, random LSTM RNNs designs are generated as all pheromones on edges are set to 1. The workers then trained the LSTM RNN on different flight data records using the backpropagation algorithm and the resulting fitness (mean squared 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 edges 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 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.

5.2 Comparison to Traditional Methods

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 [10, 17], especially when using the back propagation through time algorithm [25] 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 10 seconds in the future, the output and error values will not be known for an additional 10 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.

5.2.1 Nonlinear Output Error (NOE) Inputs Neural Network: The structure of the NOE network is depicted in Figure 4. 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.

5.2.2 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 [16], 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 5 was used. However, the output of the cost functions in the training iterations of this implementation froze at a constant value,

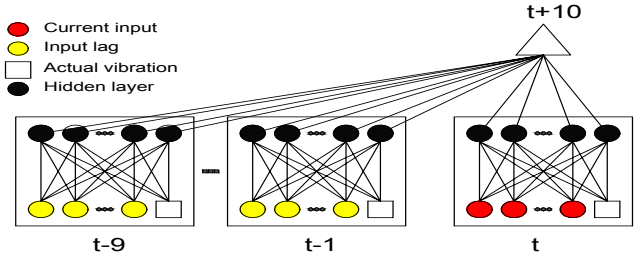


Figure 4: Nonlinear Output Error inputs neural network. This network was updated to utilize 10 seconds of input data.

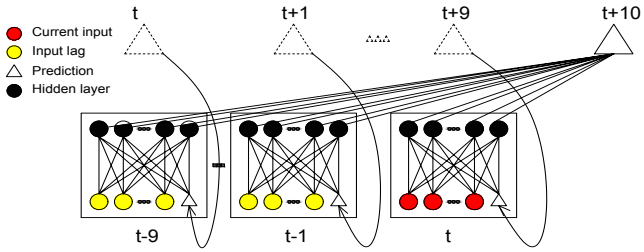


Figure 5: Nonlinear AutoRegressive with eXogenous inputs neural network. This network was updated to utilize 10 seconds of input data, along with the previous 10 predicted output values.

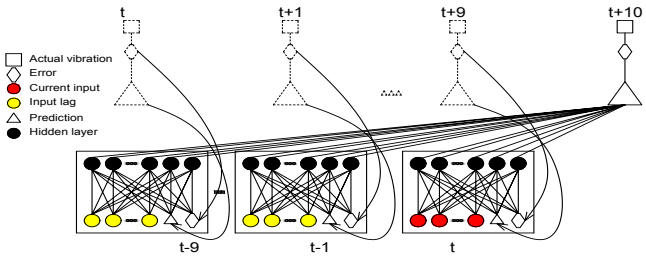


Figure 6: Nonlinear Box-Jenkins inputs neural network. This network was updated to utilize 10 seconds of input data, along with the future output and error values. Due to requiring future knowledge, it is not possible to utilize this network in an online fashion.

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.

5.2.3 Nonlinear Box-Jenkins (NBJ) Inputs Neural Network: The structure of the NBJ is depicted in Figure 6. 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 delt 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.

Table 1: K-Fold Cross Validation Results

	Prediction Errors (MAE)				
	LSTM	**NOE**	**NARX**	**NBJ**	**ACO**
Subsample 1	8.34%	10.6%	8.13%	8.40%	7.80%
Subsample 2	4.05%	6.96%	6.08%	7.34%	3.70%
Subsample 3	6.76%	16.8%	11.2%	13.6%	3.49%
Mean	**0.0638**	**0.1145**	**0.0847**	**0.0977**	**0.0501**
Std. Dev.	**0.0217**	**0.0497**	**0.0258**	**0.0333**	**0.0245**

5.3 Error Function

For all the networks studied in this work, Mean Squared Error (MSE) (shown in Equation 1) 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 2) 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.

$$Error = \frac{0.5 \times \sum(Actual\ Vib - Predicted\ Vib)^2}{Testing\ Seconds} \quad (1)$$

$$Error = \frac{\sum[ABS(Actual\ Vib - Predicted\ Vib)]}{Testing\ Seconds} \quad (2)$$

5.4 Machine Specifications

Python's Theano Library [23] was used to implement the neural networks and MPI for Python [3] and was used to run the ACO optimization on a high performance computing cluster. The cluster's operating system was Red Hat Enterprise Linux (RHEL) 7.2, and had 31 nodes, each with 8 cores (248 cores in total) and 64GBs RAM (1948 GB in total). The interconnect was 10 gigabit (GB) InfiniBand.

6 RESULTS

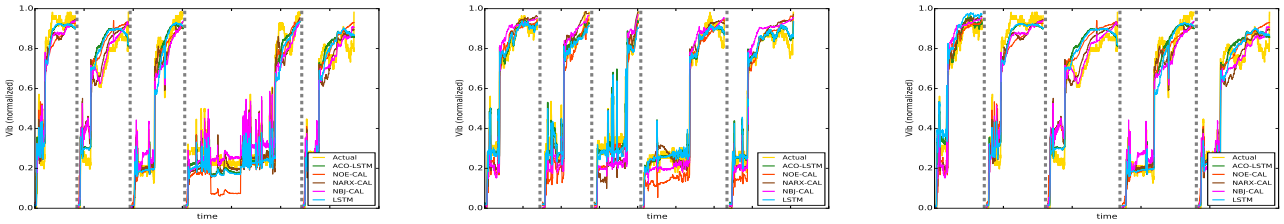
The ACO algorithm was run for 1000 iterations using 200 ants. The networks were allowed to train for 575 epochs to learn and for the error curve to flatten. The minimum value for the pheromones were 1 and the maximum was 20. The population size was equal to number number of iterations in the ACO process, *i.e.*, the population size was also 1000. Each run took approximately 4 days.

A dataset of 57 flights was divided into 3 subsamples, each consisting of 19 flights. The subsamples were used to cross validate the results by examining combations utilizing two of the subsamples as the training data set and the third as the testing set. Subsamples 1, 2 and 3 consisted of 23,371, 31,207 and 25,011 seconds of flight data, respectively.

These subsamples were used to train the NOE, NARX, NBH, base architecture and the ACO optimized architecture. Figures 7 shows predictions for the different models over a selection of test flights, and Figure 8 shows predictions an single uncompressed (higher resolution) test flight. Table 1 compares these models to the base architecture (LSTM) and the ACO optimized architecture (ACO).

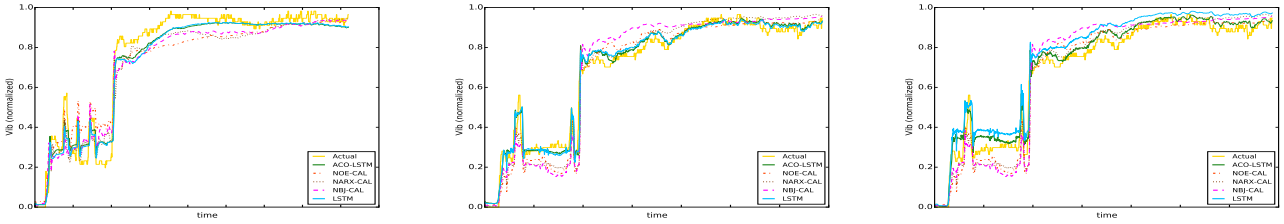
6.1 NOE, NARX, and NBJ Results

Somewhat expectedly, the NOE model performed the worst with with a mean error of 11.45% ($\sigma = 0.0497$). The NBJ model performed better than the NOE model with a mean error of 9.77% ($\sigma = 0.0333$),



(a) Test Subsample 1 (b) Test Subsample 2 (c) Test Subsample 3

Figure 7: Results for the RNNs trained on the different K-fold K-fold cross validation subsamples predicting vibration ten seconds in the future for a selection of 5 flights from the unseen subsample. Each flight is divided by a dashed line.



(a) Test Subsample 1 (b) Test Subsample 2 (c) Test Subsample 3

Figure 8: Results for the RNNs trained on the different K-fold cross validation subsamples predicting vibration ten seconds in the future for a different individual flight from the unseen subsample.

however the NARX model better than the previous two models with a mean error of 8.47 % (σ = 0.0258). 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 differences 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.

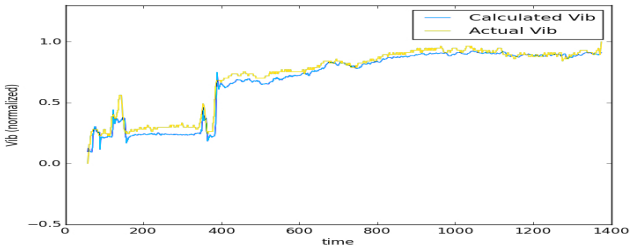
6.2 Base Architecture Revisited

The base architecture was trained utilizing the three subsamples to validate the results. The mean error for each of the three subsamples (using the other two as training data) was 6.38% (σ = 0.0217).

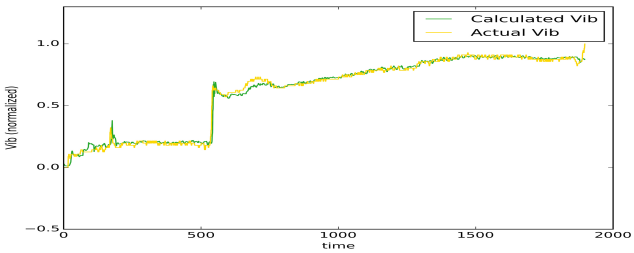
6.3 Ant Colony Optimized Architecture

When ACO optimization was used to find the optimal connections to use in the base architecture RNN, the best evolved with ACO showed an improvement of 1.37% for predictions 10 seconds in the future, reducing prediction error from 6.38% to 5.01% compared to the architecture's performance before ACO. Figure 9a provides an example of the improvement in predictions on a single test flight, before and after the ACO optimization.

Figure 10 provides an example of how the M1 cells are updated with these connections. For clarity, Figure 10b shows the differences between the M1 cells before and after ACO optimization. Figure 10a is simply a LSTM cell "M1" that have its gates' meshes (shown in Figure 10b, Up) substituted with the ACO meshes (shown in Figure 10b, Down). "M2" did not change from its original topology as shown in Figure 1 since all the elements in *mesh_2* after the



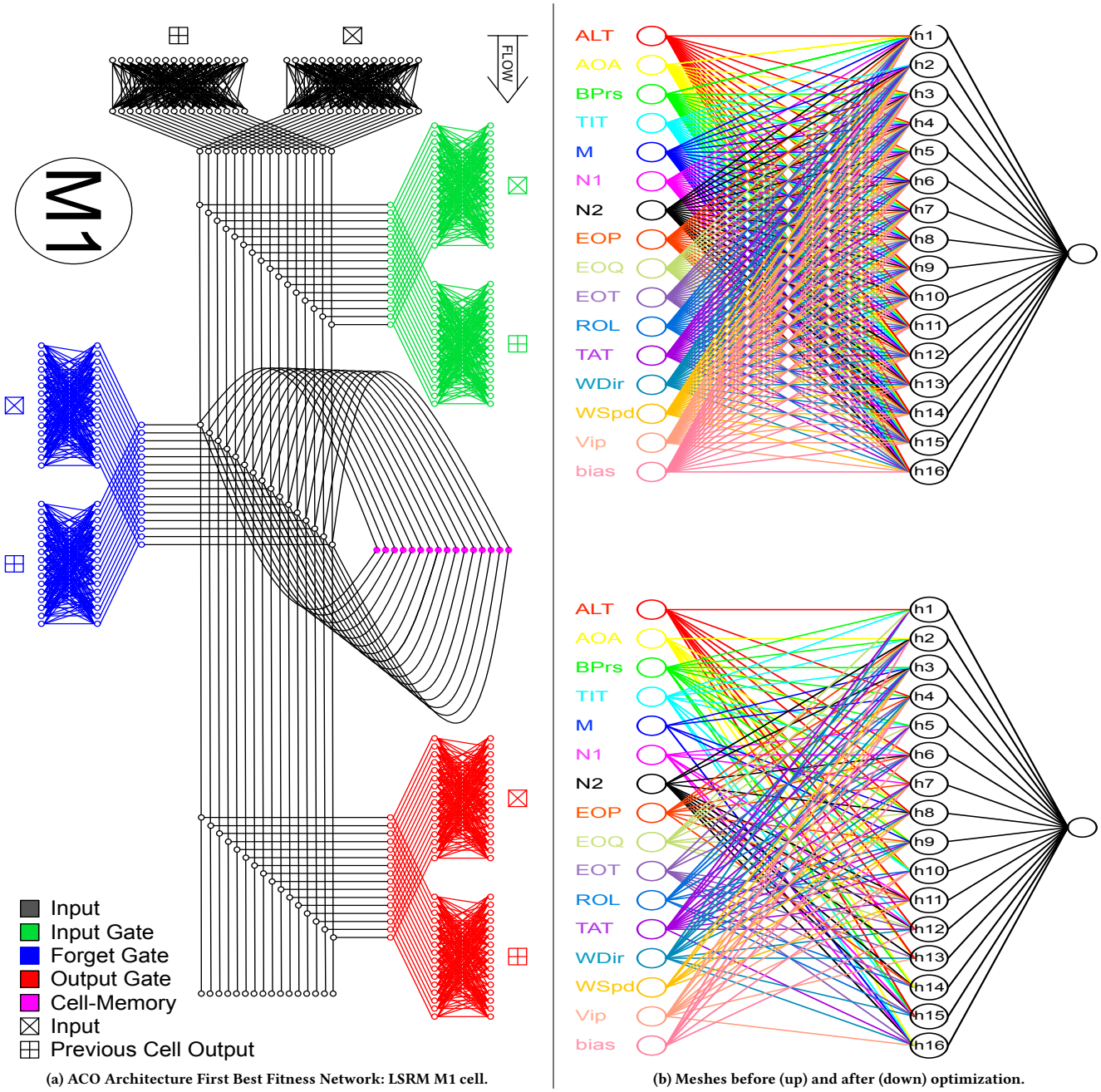
(a) Unoptimized: single test flight



(b) Optimized: single test flight

Figure 9: Plotted results for predicting ten seconds in the future.

optimization remained active. 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.



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

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

Figure 10: An example of the M1 cell before and after optimization.

Returning to an initial question of how the number of the connections in the network affects the soundness of the results, Table 2 shows the fitnesses, number of M1 and M2 connections and total number of weights for the top 30 evolved LSTM RNNs. Comparing these to the number of connetions in the base architecture, which had 21,170 weights, ACO reduced the total number of weights in these top 30 architectures by 42% to 45%.

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. The evolved structures in the M1 cell connections (see Figure 10b) were sparse; however no inputs ended up being fully removed, which was discussed a potential means for improving predictions in Section 1. This is a

Table 2: 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	0.037161	139	16	11810

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).

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. Future work will evolve each mesh in the LSTM RNN independently using ACO.

In addition, all connections of the structure shown in Figure 1 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 remove unused cells the LSTM RNN.

As this work shares some similarities with dropout strategies for recurrent neural networks [27], it is worth investigating if 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.

REFERENCES

[1] M Annunziato, M Lucchetti, and S Pizzuti. 2002. Adaptive Systems and Evolutionary Neural Networks: a Survey. *Proc. EUNITE02, Albufeira, Portugal* (2002).

[2] Edward Choi, Mohammad Taha Bahadori, and Jimeng Sun. 2015. Doctor AI: Predicting Clinical Events via Recurrent Neural Networks. *arXiv preprint arXiv:1511.05942* (2015).

[3] Lisandro Dalcín, Rodrigo Paz, Mario Storti, and Jorge DâĂŹElía. 2008. MPI for Python: Performance improvements and MPI-2 extensions. *J. Parallel and Distrib. Comput.* 68, 5 (2008), 655–662.

[4] Travis Desell, Dave Anderson, Malik Magdon-Ismail, Boleslaw Szymanski Heidi Newberg, and Carlos Varela. 2010. An Analysis of Massively Distributed Evolutionary Algorithms. In *The 2010 IEEE congress on evolutionary computation (IEEE CEC 2010)*. Barcelona, Spain.

[5] Travis Desell, Sophine Clachar, James Higgins, and Brandon Wild. 2015. Evolving Deep Recurrent Neural Networks Using Ant Colony Optimization. In *European Conference on Evolutionary Computation in Combinatorial Optimization*. Springer, 86–98.

[6] Luca Di Persio and Oleksandr Honchar. Artificial neural networks approach to the forecast of stock market price movements. (????).

[7] AbdElRahman ElSaid, Brandon Wild, James Higgins, and Travis Desell. 2016. 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, 260–269.

[8] Martin Felder, Anton Kaifel, and Alex Graves. 2010. Wind power prediction using mixture density recurrent neural networks. In *Poster Presentation gehalten auf der European Wind Energy Conference*.

[9] Dario Floreano, Peter Dürr, and Claudio Mattiussi. 2008. Neuroevolution: from architectures to learning. *Evolutionary Intelligence* 1, 1 (2008), 47–62.

[10] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[11] Eric R Kandel, James H Schwartz, Thomas M Jessell, Steven A Siegelbaum, and A James Hudspeth. 2000. *Principles of neural science*. Vol. 4. McGraw-hill New York.

[12] Nate F Kohl. 2009. Learning in fractured problems with constructive neural network algorithms. (2009).

[13] Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. 2009. Exploring strategies for training deep neural networks. *Journal of Machine Learning Research* 10, Jan (2009), 1–40.

[14] Nijolė Maknickienė and Algirdas Maknickas. 2012. Application of neural network for forecasting of exchange rates and forex trading. In *The 7th international scientific conference" Business and Management*. 10–11.

[15] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. 2017. Evolving deep neural networks. *arXiv preprint arXiv:1703.00548* (2017).

[16] Oliver Nelles. 2013. *Nonlinear system identification: from classical approaches to neural networks and fuzzy models*. Springer Science & Business Media.

[17] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*. 1310–1318.

[18] Aditya Rawal and Risto Miikkulainen. 2016. Evolving deep lstm-based memory networks using an information maximization objective. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. ACM, 501–508.

[19] S. Hochrieter & J. Schmidhuber. 1997. Long Short Term Memory. *Neural Computation 9(8):1735-1780* (1997).

[20] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research* 15, 1 (2014), 1929–1958.

[21] Kenneth O Stanley and Risto Miikkulainen. 2002. Evolving neural networks through augmenting topologies. *Evolutionary computation* 10, 2 (2002), 99–127.

[22] Boleslaw Szymanski, Travis Desell, and Carlos Varela. 2007. 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.

[23] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688 (May 2016). http://arxiv.org/abs/1605.02688

[24] Andrew James Turner and Julian Francis Miller. 2013. 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, 213–226.

[25] Paul J Werbos. 1990. Backpropagation through time: what it does and how to do it. *Proc. IEEE* 78, 10 (1990), 1550–1560.

[26] Shimon Whiteson. 2005. Improving reinforcement learning function approximators via neuroevolution. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*. ACM, 1386–1386.

[27] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

[28] Byoung-Tak Zhang and Heinz Muhlenbein. 1993. Evolving optimal neural networks using genetic algorithms with Occam's razor. *Complex systems* 7, 3 (1993), 199–220.