On the use of LSTM networks for Predictive Maintenance in Smart Industries

Fabrizio De Vita, Dario Bruneo
Department of Engineering,
University of Messina, Italy
Email: {fdevita,dbruneo}@unime.it

***Abstract*—Aspects related to the maintenance scheduling have become a crucial problem especially in those sectors where the fault of a component can compromise the operation of the entire system, or the life of a human being. Current systems have the ability to warn only when the failure has occurred causing, in the worst case, an offline period that can cost a lot in terms of money, time, and security. Recently, new ways to address the problem have been proposed thanks to the support of machine learning techniques, with the aim to predict the Remaining Useful Life (RUL) of a system by correlating the data coming from a set of sensors attached to several components. In this paper, we present a machine learning approach by using LSTM networks in order to demonstrate that they can be considered a feasible technique to analyze the "history" of a system in order to predict the RUL. Moreover, we propose a technique for the tuning of LSTM networks hyperparameters. In order to train the models, we used a dataset provided by NASA containing a set of sensors measurements of jet engines. Finally, we show the results and make comparisons with other machine learning techniques and models we found in the literature.**

***Index Terms*—Predictive Maintenance; Deep Learning; TensorFlow; Keras; Smart Industry; Industry 4.0**

I. Introduction

During the recent years, the interest for the estimation of the equipment conditions is increasing. If we consider the definition of a system as a set of components which work together to execute one or more tasks, it is easy to understand that the "health" of a system (and its reliability) depends on the working conditions of its components. Sometimes, the failure of a component can affect the entire operation causing a period during which the system is totally offline and requests maintenance. In some contexts, the unavailability of a system could cause several problems which may include the loss of human lives in the worst case.

Usually, these systems work in a way that consists in monitoring all the data coming from a set of sensors and sending an alarm signal when one or more core elements have a fault or do not work properly. The main problem in this case is given by the fact that the system is only able to *detect* a problem and not to prevent it by just communicating it to the user when it is too late.

In Smart Industries, Predictive maintenance is one of the most used techniques to address this kind of problem since it allows to evaluate the conditions of a specific equipment, with the aim to predict the maintenance *before* the failure [1]. By using this kind of technique, it is possible to prevent unplanned maintenance which would cause offline periods that can cost a lot money for an industry. To evaluate a system condition there are a lot of techniques, but the majority of them imply the use of sensors directly mounted on the components which produce real time measurements that can be compared with the standard working condition written in the datasheet of each component, in order to check if the system is correctly working.

In such a context, the Remaining Useful Life (RUL) prediction is fundamental to deploy a good strategy which avoids unnecessary maintenance [2]. The concept of RUL defines the useful life of a component, i.e., the remaining useful period during which it will work in a proper way.

Nowadays, there is not a predominant technique to obtain the most accurate prediction, but a lot of approaches are starting to use *machine learning* in order to design a model which is able to predict the RUL. Usually the RUL is not known a priori and it can be estimated starting from the available information provided by the history of the component [3]. Due to the complex nature of some systems, the RUL estimation can be very difficult especially when the number of data coming from the sensor is too large. In such a context, the use of a machine learning approach could be useful in order to understand all the hidden correlations between the sensor measurements, allowing an accurate RUL prediction.

In this paper, we propose a machine learning approach through the use of Long Short Term Memory (LSTM) networks to analyze sensor time series sequences to estimate the RUL of turbofan engines, and we provide an analysis where we show how the LSTM performance changes when varying its internal hyperparameters. In particular, the predicting power of a model can be subject to strong fluctuations when architecture changes even when we use the same dataset to train it.

The purpose of the paper is twofold. First of all, we perform an analysis which shows how the hyperparameters can affect the model with the final goal to discover possible correlations between them and the model performance. To do that, we used a dataset containing the full history of a set of engines until their fault provided by NASA [4] that can be considered a standard to train and test models for the RUL prediction. Moreover, starting from the above analysis, we design an LTSM network that is able to outperform the other solutions present in the literature. The paper is structured as follows: Section

978-1-7281-1689-1/19/$31.00 ©2019 IEEE
DOI 10.1109/SMARTCOMP.2019.00059

II introduces the concept of predictive maintenance, explains how LSTM networks work, and provides a description of the dataset we used to train our models, Section III provides more details about the methodology we used to train and test the models we designed, Section IV shows the results we obtained, finally Section V closes the paper with the conclusions also giving some details about future works.

II. Predictive Maintenance

The main goal of predictive maintenance is to make an accurate estimation of the RUL of a system. Traditional systems are only able to warn the user when it is too late and the fault occurred causing an unpredictable offline period during which the system cannot operate properly with a consequent waste of time and resources.

In such a context, we introduce the concept of RUL and give a better definition of it. Given a system, the RUL can be considered as an index that synthesizes its life status or of a part of it. In general, as we mentioned in Section I, the RUL is something that is not known a priori, but it is estimated starting from the known information of a specific component in combination with a monitoring process during its healthy working condition [3]. From a statistical point of view, the RUL can be expressed as $f(x_t|Y_t)$ where x_t is a random variable associated to the RUL at time t and Y_t represents the history of the system up to time t [3].

In order to evaluate the conditions of a system, the predictive maintenance approach employs sensors of different nature (e.g., temperature, vibration, noise) attached to the core components whose fault would compromise the entire system operation. In this sense, predictive maintenance analyzes the history of a system in terms of the measurements gathered by the sensors which are spread among the components, with the aim to extract a "fault pattern" which can be exploited to plan an optimal maintenance strategy and thus reducing the offline periods [5].

However, when the systems start to be too complex or the number of sensor measurements to manage is too large, it could be difficult to estimate a fault, for this reason in the recent years machine learning techniques are becoming more and more used to predict the working conditions of a component. Authors in [6], propose several machine learning approaches like support vector machines (SVMs), decision trees (DT), Random Forests (RF), and others showing which technique performs best in predicting the RUL of turbofan engines.

The work presented by authors in [5] demonstrates that LSTMs can be considered a valid solution that outperforms other techniques like Naive Bayesian regression. The key idea behind the use of the machine learning in this sense is to design a model which is able to learn the "life"distribution of a system from the experience provided by its history and understand which are the faulty conditions.

Even if at the moment it does not exist a best technique, LSTM networks turned out to be a winning choice thanks to their ability to keep in memory long time series dependencies extracting at the same time complex correlations between data which provide very useful information in determining the RUL or more in general the state of the system.

A. LSTM Networks

In this paper, we propose a model based on an LSTM network. This kind of networks basically are Recurrent Neural Networks (RNNs) which use an efficient gradient based algorithm to keep constant the error avoiding its explosion or vanishment [7]. The main advantage with respect to traditional RNNs is given by the fact that LSTM networks are able to keep in memory time dependencies between inputs for a longer period. One of the main drawbacks regarding the RNNs in fact is their ability to keep track only of recent past dependencies due to the gradient exponential decay also known as *vanishing gradient problem* that does not allow to perform the backpropagation algorithm through time (BPTT) for sequences that are too long [7].

Fig. 1 shows how the LSTM cell is internally organized. It has three gates (namely, the input gate, the output gate, and the forget gate) ruled by the following equations.

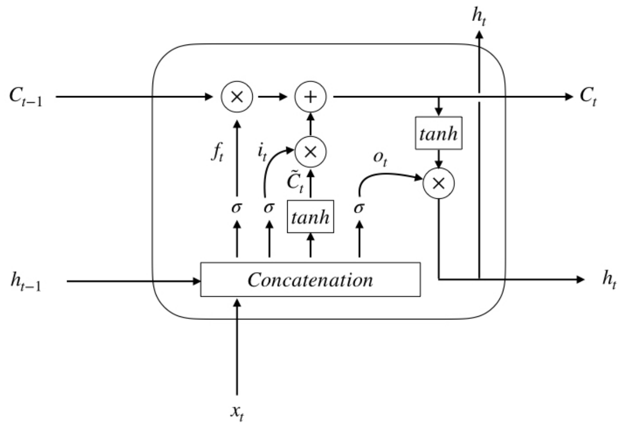


Fig. 1: LSTM cell.

$$f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f) \tag{1}$$

$$i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i) \tag{2}$$

$$\widetilde{C_t} = tanh(W_C * [h_{t-1}, x_t] + b_C) \tag{3}$$

$$C_t = f_t \circ C_{t-1} + i_t \circ \widetilde{C_t} \tag{4}$$

$$o_t = \sigma(W_o * [h_{t-1}, x_t] + b_o) \tag{5}$$

$$h_t = o_t \circ tanh(C_t) \tag{6}$$

where $\circ$ is the symbol for the element wise product, f_t,i_t,o_t are the *forget gate*, the *input gate*, and the *output gate*, and W_f, W_i, W_C, W_o are the weight matrices of the forget gate, input gate, cell state, and output gate.

When the data flows inside the LSTM cell, the first step consists in deciding what information is necessary to keep and what is not. This decision is made through eq. (1) which takes as input the data array x_t at the time instant *t* together with the output h_{t-1} coming from the time instant *t-1*. In such a context, x_t is an array containing for each time instant t a collection of data representing the input features of the system (e.g., sensor data). The result obtained is then passed as input to the sigmoid activation function σ which returns a number between 0 and 1 for each element of the cell state C_{t-1} where 0 means that the element can be forgotten while 1 means that the element has to be kept over time. The second step involves the input gate and allows the LSTM to establish which of the new information to store. This is done by executing two sub steps, the first one consists in passing the new information to a sigmoid activation function (eq. (2)) which returns a value between 0 and 1 just like the previous step, where 1 means that the element has to be updated while 0 means that the element has not to be updated. The second substep allows to compute the new state values through eq. (3) where the hyperbolic tangent *tanh* is an activation function adopted mainly for two reasons: *i)* it normalizes the output between -1 and 1; *ii)* it has a derivative which is particularly fitting to address the vanishing gradient problem which affects traditional RNNs. At the end of these substeps, the new LSTM cell state can be updated through eq. (4) which will be a combination of the new values computed in the current timestep (eq. 3), and the values coming from the previous one. Finally the last step allows the generation of the LSTM output through eqs. (5) and (6) that select which part of the new LSTM state to output.

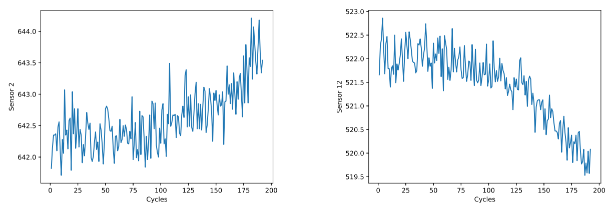
B. Dataset Description

In this subsection, we provide more details about the dataset we used to train and test our models. As already said in Section I, our goal is to predict the RUL and in particular we are interested to estimate it for a set of industrial engines with the goal of reducing the offline periods due to maintenance.

We faced this problem as a supervised regression problem, this means that the output we generate is a number representing the RUL. Since we used a supervised approach we needed a labeled dataset containing the entire “history” of a set of engines until their failure. Due to the complexity in replication of such a scenario which would require the usage of a large set of engines until their fault, we opted for a dataset provided by NASA through C-MAPSS tool which simulates the engine degradation. The dataset consists of a multiple multivariate timeseries composed of three operational settings that describe the environment conditions in which the engines are working and 21 sensors measurements with in addition the engine *id* and time *cycle* that can be used to discriminate over the “life” of each engine.

Fig. 2 shows a typical “fault pattern” of an engine for two different sensors. As we can observe, both plots 2a and 2b depicts an evident trend that leads the engine to the failure.

Table I gives a better view of the dataset organization. Each row represents a time cycle of an engine and consists of a series of sensors readings for the generic engine identified by its ID.



(a) Sensor 1 (b) Sensor 12

Fig. 2: Sensors readings of engine 1 until its fault.

The dataset comes in four versions which differ for the operational settings, and already split in train and test sets [1]. In particular, we worked with the first dataset which contains the history of a set of 100 engines and where the train and test sets can be assumed as matrices where the number of rows (samples) is respectively 20,631 and 13,096 and the number of columns (features) is 26. Regarding the test label dataset, it is just a column vector containing the ground truth RUL for each one of the engines.

Since the dataset comes with only the test labels without the training ones, it has been necessary to compute them, in order to provide during the training process an input sequence of measurements together with the corresponding RUL. As already said, each row of the dataset represents a cycle of an engine; since the train set contains the full history of each engine until the fault and the RUL can be defined as the remaining number of cycles before the engine stops to work, it is possible to compute the RUL by subtracting to the last value of cycle registered for the generic engine the value of the *Cycle* feature assumed at the *i-th* cycle. For a better understanding, we provide the equation we used for the computation:

$$RUL_{ith} = last\_cycle - Cycle_{ith} \tag{7}$$

where $last\_cycle$ is the last value taken by the *Cycle* feature of an engine.

Moreover, we normalized the dataset using the min max normalization with values between -1 and 1, this is typically done when the feature values have different scales which leads to giving more importance to features with larger values thus leading the trained model to wrong assumptions that can cause an increment of the error.

After the normalization, we started a manual features extraction process in order to provide a first model “lightening”. In particular, we decided to drop the feature column relative to the engine id because it was not informative for the LSTM model due to the fact that the engine life history and the RUL are not affected by its ID which is just a number to identify it. At the end of this process, the dataset was ready to be fed to the LSTM network for the training phase.

III. Adopted Methodology

In this section, we provide more details regarding the methodology we adopted to analyze the models we designed

Turbofan Engine Degradation Simulation dataset									
$Engine\ ID$	$Cycle$	$Setting\ 1$	$Setting\ 2$	$Setting\ 3$	$Sensor\ 1$	$Sensor\ 2$	$Sensor\ 3$	...	$Sensor\ 21$
1	1	0.0023	0.0003	100	518.67	643.02	1585.29	...	23.3735
1	2	-0.0027	-0.0003	100	518.67	641.71	1588.45	...	23.3916
...	...	...	...	...	...	...	...	...	...
1	31	-0.006	0.0004	100	518.67	642.58	1581.22	...	23.3552
...	...	...	...	...	...	...	...	...	...
100	198	0.0013	0.0003	100	518.67	642.95	1601.62	...	23.1855

TABLE I: NASA Dataset.

by varying different hyperparameters. Since the hyperparameter space is too large, it would be impossible to analyze all of them, for this reason, we focused on a subset.

From our experience, we noticed that the obtained results can be very different when varying the model architecture so the base idea of such an analysis is to show how the model performance, in terms of Root Mean Squared Error (RMSE), changes over a different set of hyperparameters. This idea comes from the fact that in the literature usually authors propose a model and present the obtained results without any insight on the hyperparameter settings, conversely, our purpose for this paper is to make comparisons between different models with the aim to select a subset of hyperparameters as a starting point for the design of model with a good level of accuracy.

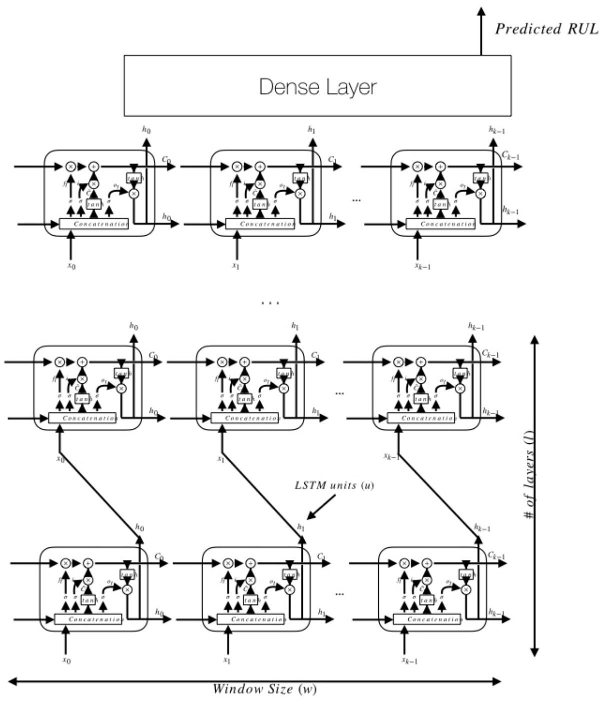


Fig. 3: Model architecture.

As usually happens with the "traditional" feed forward neural networks, also in the case of RNNs it can be necessary to design a model with several layers between the input and the output. This is typically done when the number of features is large and just one layer is not enough to allow the neural network to learn the complex relationship between the input and the output. In such a context, deep learning is a valid solution to address such a kind of problem [8]. The concept behind this technique consists in adding several layers connected in cascade with the aim to improve the predicting power of a model thanks to the features extraction process which allows to get rid of all those features which are redundant or not useful for the output estimation.

From an architectural point of view, a LSTM cell acts like a RNN, this means that we can think to "unroll" it in order to see all the timesteps it stores internally. Figure 3 shows an unrolled LSTM which takes as input different inputs delayed over time together with the cell state and the output of the previous timestep except for the the first one (t = 0) where in this case the initial state is set to zero.

When two LSTM layers are connected in cascade this means that if we unroll the LSTM cells, for each time step, the *l-th+1* layer will receive as input a hidden vector computed in the previous *l-th* layer through the eqs. (5) and (6) where W_o is a matrix whose dimension depends on the "width" of the unrolled LSTM cell and on the number of its internal units. Since we are training our model to perform a regression, at the end of the architecture is necessary to use a *dense layer* to generate as output only one number which in this context is represented by the RUL prediction given in input a sequence of sensors measurements.

With reference to Fig. 3, let us introduce the following hyperparameters: the number of layers (l), the window size (w) and the LSTM units (u), where the first one defines the "depth" of the model architecture that allows to extract more and more complex features as long as we go deep, the second one defines the number of timesteps that are kept in memory by the single LSTM cell in order to learn the time dependencies inside a sequence, and the last one defines the output dimension of the LSTM cell or in other words it defines the dimension of the hidden vector that we mentioned before which will have a u dimension. With respect to traditional feed forward (FF) networks, the number of units can be seen as the neurons inside a hidden layer. In this sense, the number of neurons in the hidden layer specifies the output dimension of the layer itself, but also its learning power in fact, by adding neurons in a network, we are increasing the number of parameters that can be learned by the network.

In particular, we started by choosing three sets of discrete values for each one of the hyperparameters, after that, we iteratively fixed one of them by changing the remaining two.

At the end of this procedure, we trained a number of models which is equal to the cardinality of the Cartesian product which can be computed by multiplying the single cardinalities of each set. By doing so, we were able to analyze the RMSE variations when changing the hyperparameters values and understand which are the elements that affects it mostly. The result of such an analysis is very important because it allows to have a better understanding on how this kind of models work and how we can improve them by changing their configuration.

IV. Results

In this section, we present the results obtained from the models designed by adopting the methodology we described in the previous section and we compare them with other machine learning models (i.e., SVMs and DNNs). As already said, we focused our analysis on three specific hyperparameter sets and we tested several models with the aim to discover possible correlations with RMSE. In particular, the RMSE is computed as follows:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(\widehat{y_i} - y_i)^2} \tag{8}$$

where $\widehat{y_i}$ and y_i are respectively the value predicted by the model and the ground truth provided by the test set of the dataset of the generic *i-th* input sample, and *n* is the number of samples of the test set. In such a context, we define another error index that expresses the mean percentage error of the model as follows:

$$Error\% = \frac{1}{n}\sum_{i=0}^{n}\frac{|\widehat{y_i} - y_i|}{y_i} \tag{9}$$

where $\widehat{y_i}$,y_i, and n are equal to the values we defined in eq. (8).

In particular, the RMSE is a statistical index which provides a generic estimation of the error by measuring the mean "distance" between the prediction and its ground truth, conversely, the percentage error provides a better view on the "impact" of error that the model makes on each sample.

LSTM parameters		
$Window\ size\ (w)$	$No.\ of\ units\ (u)$	$No.\ of\ layers\ (l)$
30	10	1
40	20	2
50	30	3
60	40	4
70	50	5
80	60	6
90	70	7

$Learning\ rate$	0.001
$Training\ epochs$	500
$Dropout$	$[0.2, 0.5]$

TABLE II: LSTM network parameters.

Table II shows the sets of values we fixed for the LSTM network design process together with the values of the learning rate, the training epochs, and the dropout. In particular we fixed a total number of seven values for each set and applying the methodology we presented in Section III we obtained a total number of $\|Window\ size\| \cdot \|Number\ of\ units\| \cdot \|Number\ of\ layers\|$ models which in this case is equal to 343. In order to train the models, we used Keras [9] a high level API written in Python that runs on top of very powerful machine learning frameworks like Tensorflow, Theano, and Caffe [10], [11], [12].

In order to train such a huge amount of models, we installed Keras on several machines and launched in parallel the train processes with different configurations. At the end of this procedure we plotted in a 3d space the RMSE trend by fixing each time one hyperparameter and varying the other two. For the sake of simplicity, we show in this paper only a part of them focusing in particular on those ones which are informative and present a visible trend.

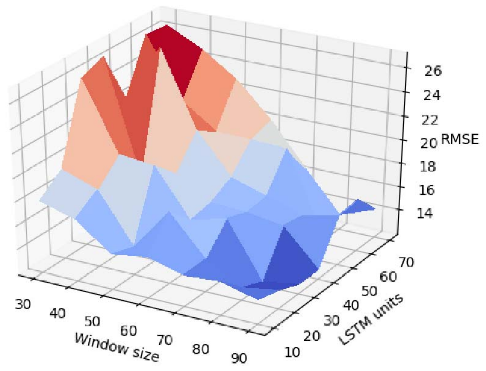


Fig. 4: RMSE plot obtained by fixing the number of layer to 2.

Fig. 4 shows the RMSE trend by fixing the number of layers to two and varying the window size and the number of LSTM units. The plot puts in evidence that the RMSE tends to decrease when we increase the windows size, and in general this is true for every number of LSTM units we decide to use. In particular, the RMSE reaches the lowest values by setting the number of LSTM units to 30 and 40. This is due to the fact that by increasing the window size, we are are giving to the LSTM model the possibility to store internally larger sequences and to learn possible correlations between "distant" timesteps. The effect we obtain as shown in the Figures we already mentioned is in both cases the reduction of the error.

Fig. 5 depicts how RMSE changes when we fix the number of LSTM units to 20 and changing the other parameters. Except for some peak which means that the network has overfitted, the overall trend is very similar to the previous plot with the lowest value in the bottom part which corresponds to the largest window size with a number of layers equal to 3. As we expected, the larger the window the better the LSTM learns

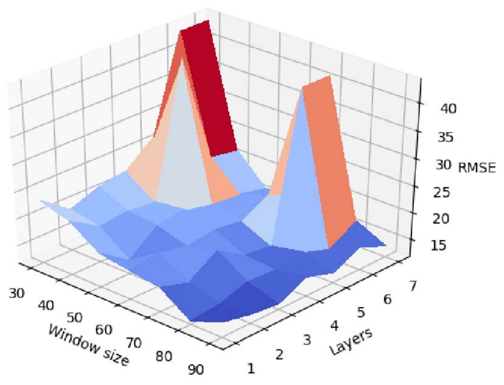


Fig. 5: RMSE plot obtained by fixing the number of units to 20.

the time dependencies in a sequence of sensors measurements.

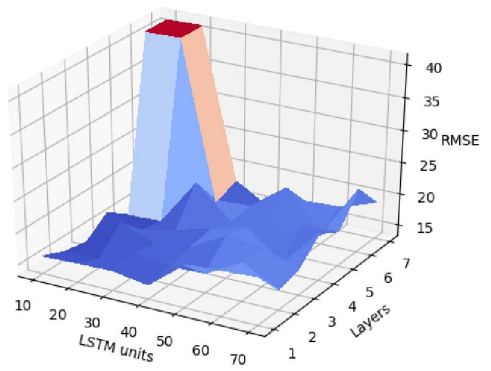


Fig. 6: RMSE plot obtained by fixing the window size to 80.

With reference to Fig. 6, in this case what we can notice is that the RMSE reaches the lowest values when layers are between 2 and 3 just like the plot of Fig. 5.

By combining the information that we extracted from all the measurements we computed, our goal was to design a new model in order to improve the performance thus reducing the error. In particular, from the plots we can observe that a larger window size improves the RMSE, and this is obtained by fixing the number of layers between two and three. Thanks to these information we were able to merge and use them to design a new model. Table III shows the configuration of the LSTM network we designed combining the information we derived from the experiments. We fixed the window size to 90 and we tried different configurations with two and three layers; from our tests, a model with three layers has always outperformed the model with two in terms of RMSE and percentage error.

Moreover, we decided to change the architecture by changing the number of LSTM units of each layer instead of using a fixed one. In particular, we fixed a number of 40 units for the first layer, 15 units for the second and the third. The idea to create a topology with a variable number of units which decreases going towards the output layer is a very common choice in deep learning architectures.

We fixed the learning rate to 0.001 with *RMSProp* optimizer which resulted a good choice from our tests and the maximum number of training epochs to 500 on which we applied an early stopping technique to prevent an overtraining of the model. Finally, we used a dropout for each layer with a 0.2 drop rate for the first layer and 0.5 for the others to prevent overfit.

LSTM model architecture	
$Window\ size$	90
$No.\ of\ layers$	3
$No.\ of\ units$	$[40, 15, 15]$
$Learning\ rate$	0.001
$Training\ epochs$	500
$Dropout$	$[0.2, 0.5, 0.5]$
$Optimizer$	$RMSProp$
$RMSE$	11.42
$Error\%$	0.12

TABLE III: LSTM network configuration.

Fig. 9 depicts a comparison between the RUL predicted by the LSTM model for each one of the engines and the correspondent ground truth. The results showed by the plot demonstrate that the model was trained correctly and in particular it has been able to understand the “fault pattern” for the most part of the test set.

For sake of simplicity, Fig. 8 provides a different view of the model performance on the test data showing the differences between the predictions made and the ground truth values where the generic *i-th* difference has been computed as:

$$d_i = \widehat{y_i} - y_i \tag{10}$$

In this sense, a value higher than zero means that our model made an optimistic prediction while a value lower than zero means a pessimistic prediction. Nevertheless, the majority of them presents a difference near to zero meaning that the model is able to correctly predict the RUL with a very good level of accuracy.

As we can observe for both the Figures, given a test set of 100 engines we can take into account 74 of them because by fixing the window size to 90 timesteps, this implies that only those engines in the test set which have a longer “history” can be considered.

A. Comparison with other techniques

In this subsection in order to demonstrate the effectiveness of our approach, we provide the results we obtained by performing the training process with other machine learning algorithms. In particular, we compared our approach with

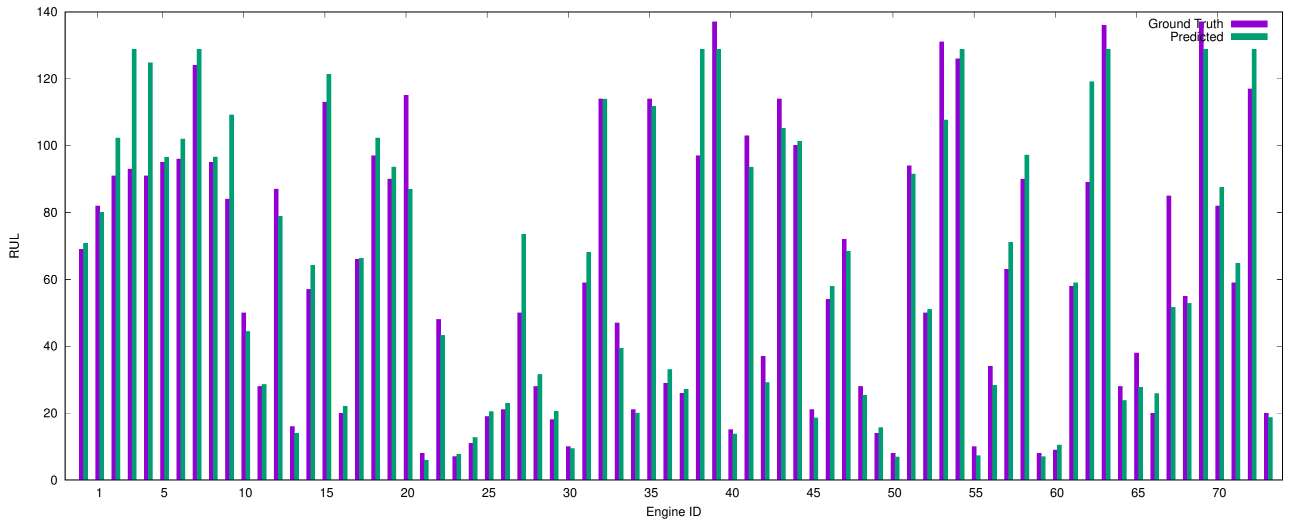


Fig. 7: Plot which shows the RUL predicted by the LSTM and the correspondent ground truth.

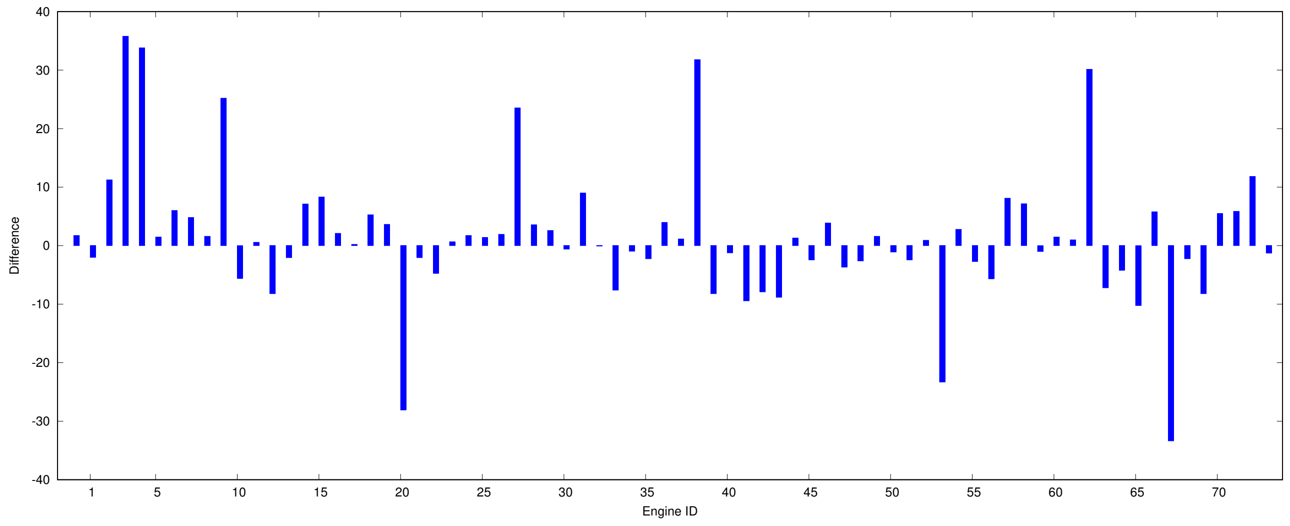


Fig. 8: Plot which shows the difference between the predicted values and the real ones.

SVMs and DNNs in order to demonstrate that LSTM networks are able to better understand the relationship between the history of the engine described by the sensor measurements and the RUL.

Regarding the DNN, we designed a network with five hidden layers between the input and the output. Also in this case, we used a topology where the number of neurons decreases going towards the output layer and in particular we fixed the number of neurons for the first hidden layer to 25 and 15 for the remaining ones with Rectified Linear Unit (ReLU) as activation function. Moreover, we used a regularization technique by introducing a penalty term in order to prevent overfit and we applied early stopping to avoid the model overtraining leaving the learning rate and the training epochs equal to the values we used for the LSTM network. Finally, we used Adam optimizer which resulted to be the best in terms of training performance.

For the SVM algorithm, we used a radial basis function kernel (RBF) which is one of the most used, a penalty term $C = 1$ which fixes the tolerance of the error that can be committed by the algorithm for each training sample and we set the number of training epochs to *auto* which means that the algorithm stops only when the solver finds the solution.

Tables IV and V provide a better view of the three approaches we tested and demonstrate that the LSTM outperforms the other two; such a result is mainly due to its ability to process input sequences and store the useful information over time inside his memory which allows it to understand possibles time correlations between them. Conversely, the SVMs and DNNs analyze each sample singularly as it is without considering the time correlations with the others thus resulting in a worse RUL prediction.

Fig. 9 depicts the comparison between the model we designed and other machine learning approaches including other results we found in the literature. In particular, we reached a RMSE score equal to 11.42 and a percentage error equal to 0.12 which resulted in the lowest value obtained if compared with the 343 models we designed during the analysis and the other machine learning models we presented in this section (i.e., SVMs and DNNs). If compared with the works presented in [5] and in [6], we were able to improve the RMSE by reducing it of about 35% and 54% respectively.

These results confirm that LSTM networks are a promising approach for predictive maintenance in smart industries and that a careful tuning of the network architecture and of the corresponding hyperparameters is needed in order to obtain good results.

SVM configuration	
$Kernel$	RBF
$Penalty\ term\ (C)$	1
$Maximum\ training\ epochs$	$auto$
$RMSE$	51.54
$Error\%$	2.18

TABLE IV: SVM configuration.

DNN model architecture	
$No.\ of\ layers$	5
$No.\ of\ units$	$[25, 15, 15, 15, 15]$
$Activation\ function$	$ReLU$
$Learning\ rate$	0.001
$Training\ epochs$	500
$Regularization\ term$	0.01
$Optimizer$	$Adam$
$RMSE$	41.77
$Error\%$	1.43

TABLE V: DNN network configuration.

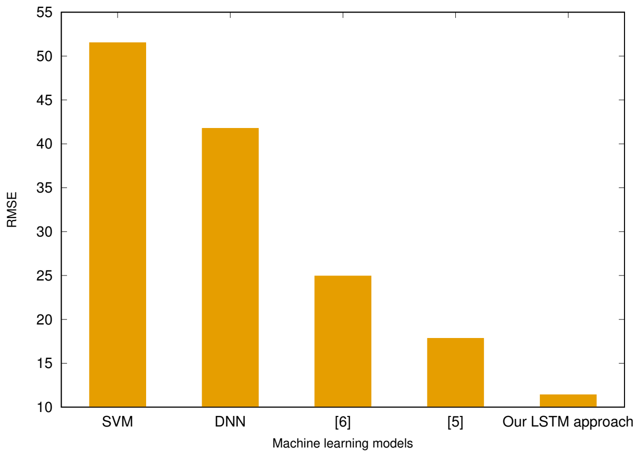


Fig. 9: Plot which shows a comparison with other techniques.

V. Conclusions

In this paper, we presented a machine learning approach for the predictive maintenance on a set of engines through the RUL estimation. We proposed an analysis showing how the hyperparameters of a LSTM network can affect the RMSE with the aim to have a better understanding on how these models work and how we can improve them. Using the information we derived from this analysis, we were able to design a new LSTM model which resulted to be the best in terms of RMSE and percentage error if compared with models we trained during the analysis and other machine learning approaches such as: SVMs and DNNs. The results obtained are encouraging, in particular our model outperforms the others we found in the literature demonstrating that our approach can be considered a valid solution for the hyperparameter tuning for this kind of models. Future works will be devoted to improve the model architecture with the aim to reduce the error and to test it in an industrial scenario to verify its capabilities in a real environment.

References

[1] O. Aydin and S. Guldamlasioglu, "Using lstm networks to predict engine condition on large scale data processing framework," in *2017 4th International Conference on Electrical and Electronic Engineering (ICEEE)*, April 2017, pp. 281–285.

[2] S. Bianchi, R. Paggi, G. L. Mariotti, and F. Leccese, "Why and when must the preventive maintenance be performed?" in *2014 IEEE Metrology for Aerospace (MetroAeroSpace)*, May 2014, pp. 221–226.

[3] X. Si, W. Wang, C.-H. Hu, and D. Zhou, "Remaining useful life estimation - a review on the statistical data driven approaches," *European Journal of Operational Research*, vol. 213, pp. 1–14, 2011.

[4] A. Saxena and K. Goebel, "Turbofan engine degradation simulation data set," NASA Ames Prognostics Data Repository, NASA Ames Research Center, Moffett Field, CA, 2008. [Online]. Available: http://ti.arc.nasa.gov/project/prognostic-data-repository

[5] D. Dong, X. Li, and F. Sun, "Life prediction of jet engines based on lstm-recurrent neural networks," in *2017 Prognostics and System Health Management Conference (PHM-Harbin)*, July 2017, pp. 1–6.

[6] V. Mathew, T. Toby, V. Singh, B. M. Rao, and M. G. Kumar, "Prediction of remaining useful lifetime (rul) of turbofan engine using machine learning," in *2017 IEEE International Conference on Circuits and Systems (ICCS)*, Dec 2017, pp. 306–311.

[7] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[9] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[10] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org/

[11] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[12] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.