Evolutionary Neural Architecture Search for Remaining Useful Life Prediction

Hyunho Mo[a,b], Leonardo Lucio Custode[a], Giovanni Iacca[a,*]

[a] *Department of Information Engineering and Computer Science*
University of Trento
Via Sommarive 9, 38123 Trento, Italy
[b] *BlueTensor Srl,*
Via I Maggio 9, 38123 Trento, Italy

Abstract

With the advent of Industry 4.0, making accurate predictions of the remaining useful life (RUL) of industrial components has become a crucial aspect in predictive maintenance (PdM). To this aim, various Deep Neural Network (DNN) models have been proposed in the recent literature. However, while the architectures of these models have a large impact on their performance, they are usually determined empirically. To exclude the time-consuming process and the unnecessary computational cost of manually engineering these models, we present a Neural Architecture Search (NAS) technique based on an Evolutionary Algorithm (EA) applied to optimize the architecture of a DNN used to predict the RUL. The EA explores the combinatorial parameter space of a multi-head Convolutional Neural Network with Long Short Term Memory (CNN-LSTM) to search for the best architecture. In particular, our method requires minimum computational resources by making use of an early stopping policy and a history of the evaluated architectures. We dub the proposed method ENAS-PdM. To our knowledge, this is the first work where an EA-based NAS is used to optimize a CNN-LSTM architecture in the field of PdM. In our experiments, we use the well-established Commercial Modular Aero-Propulsion System Simulation (C-MAPSS) dataset from NASA. Compared to the current state-of-the-art, our method obtains better results in terms of two different metrics, RMSE and Score, when aggregating across all the C-MAPSS sub-datasets. Without aggregation, we achieve lower RMSE in 3 out of 4 sub-datasets. Our experimental results verify that the proposed method is a reliable tool for obtaining state-of-the-art RUL predictions

*Corresponding author

Email addresses: hyunho.mo@unitn.it (Hyunho Mo), leonardo.custode@unitn.it (Leonardo Lucio Custode), giovanni.iacca@unitn.it (Giovanni Iacca)

and as such it can have a strong impact in several industrial applications, especially those with limited available computing power.

Keywords: Evolutionary Algorithm, Convolutional Neural Network, Long Short Term Memory, Remaining Useful Life, C-MAPSS.

1. Introduction

Predictive Maintenance (PdM) is a new trend in research and industry to provide a benefit in terms of performance and costs of downtime. This is achieved by making better maintenance decisions based on predictions regarding the future state of physical assets. Lately, PdM has received increased attention especially for preventing unexpected failures of machines to accomplish an effective automation of industrial practices. In Industry 4.0, the Prognostics and Health Management (PHM) of industrial components is considered as a key concept for PdM [1], and Remaining Useful Life (RUL) prediction has become a mainstream element of PHM.

This prediction can be achieved by observing degradation-based measures using multiple sensors which monitor physical properties of the target machine[2]. In other words, an accurate RUL prediction for PdM requires to analyze the degradation patterns in time series collected by the sensor. Machine Learning (ML) has been used to recognize such patterns in the data. Recently, many Deep Learning (DL)-based methods have been proposed to make accurate predictions via end-to-end learning frameworks without complex feature engineering. Early work in this research area employed Convolutional Neural Networks (CNNs) [3] to estimate the RUL of components. Recurrent Neural Networks (RNNs) including Long Short Term Memory (LSTM) [4] have also been used, to consider long term dependencies in sensor input sequences, rather than convolutional features. More recently, a deep CNN (DCNN) [5] and an RNN-based deep architecture [6] has been proposed to get more precise predictions. Moreover, their combination, such as in CNN-LSTM [7, 8], proved to be promising.

Despite the promising performance of Deep Neural Networks (DNNs) in RUL prediction, these techniques largely depend on various parameters. Even researchers who are knowledgeable in both DL and PdM struggle to analyze how the parameter settings affect the performance of the available models. Furthermore, the number of parameters of the available models has been increasing because the most recent works in this field tend to employ deeper or more complex models to achieve better results. This entails that exploring such large parameter spaces to improve performance has become harder than ever. Therefore, relying on the knowledge of the experts, or empirically searching for the best parameters by trial-and-error, may not be the most efficient approaches.

In this paper, we propose to use an Evolutionary Computation (EC)-based Neural Architecture Search (NAS) method to explore a DNN parameter space and find the best model for RUL prediction. Since multi-head CNN-LSTM models [9] have shown advantages in RUL prediction, we use the high-level structure of this DNN to determine the search space explored by the Evolutionary Algorithm, and apply NAS on this search space specifically for PdM purposes. In the following, we refer to the proposed method as ENAS-PdM.

The main contributions of this work are as follows. We introduce a way of applying EC-based NAS on top of the high-level structure of a multi-head CNN-LSTM. We then develop the best architecture as a RUL predictors for PdM purposes. Our study highlights that a solution obtained on a specific sub-dataset by ENAS-PdM can be generalized to other sub-datasets. In other words, the optimized architecture performs well on all the sub-datasets. In addition, the computational cost of the evolutionary search is reduced by implementing specific techniques, so that the proposed method is suitable for industrial applications which may have limited access to large computational resources. Finally, the best architecture discovered by ENAS-PdM improves the accuracy of the RUL prediction compared to the state-of-the-art methods.

The rest of paper is structured as follows. Section 2 introduces related works in the field of NAS. In Section 3, the background concepts along with the multi-head CNN-LSTM model are introduced. The details of the ENAS-PdM algorithm are presented in Section 4. Then, Section 5 discusses the details of our experimental setup and Section 6 discusses the results of our experimentation. Finally, we provide our conclusions in Section 7.

2. Related work

The field of NAS has seen an incredible development in recent years. [10, 11, 12] make a summary (from different perspectives) of the state-of-the-art in the NAS field. One common aspect highlighted in the existing literature is that most NAS approaches are very specific to the model at hand, and focus on computer vision tasks. On the other hand, as highlighted in [10], it would be very important to apply and test NAS *outside* the computer vision domain to assess its capabilities outside fields where significant human effort has been put in engineering optimized architectures. Some common approaches to the NAS problem are described below.

Bayesian approaches make use of Bayesian Optimization (BO) to optimize the structure of DNNs. In [13] the authors introduce a new kernel that captures the relevant parameters. Kandasamy et al., in [14], introduce a distance metric that can be computed efficiently. In [15], the authors introduce a neural network kernel to perform optimization on neural networks. Other works [16, 17] focus on reducing the computational budget needed to perform Bayesian NAS. In [18], BO is used to optimize jointly both the DNN architecture and its hyper-parameters.

Reinforcement Learning (RL) approaches cast the NAS problem into a RL one. The approaches presented in [19, 20] tackle the NAS problem by using Q-learning. In [21], the authors speed up the search process described in [19] by predicting the performance using a predictor. In [22] the authors use a RNN to produce NN architectures. Cai et al. [23] use a similar approach but improving its efficiency. In [24], the authors allow for intra-layer modifications to further improve the efficiency of the process. In [25], the authors use an RNN to produce a convolutional cell. Pham et al. [26] aim to increase the velocity of the search process by making all the neural networks share their parameters. In [27], RL is used to reduce the size of a NN while keeping satisfactory performance.

Gradient-based techniques perform NAS by creating a differentiable version of the NAS problem and optimizing the parameters via gradient descent. In [28, 29], the authors make use of a NN to predict the validation accuracy of other NNs. In [30], graph hyper-networks are used to considerably reduce the NAS runtime. In [31] the authors perform a continuous relaxation of the NAS problem that, by using backpropagation, "chooses" only the relevant parts of the starting NN. Xu et al., in [32], extend [31] to reduce the redundancy in the search space, thus increasing the efficiency of the process. Xie et al. [33] cast the problem as the continuous optimization problem of a basic cell. In [34], the authors propose an efficient approach to NAS by reducing the memory footprint. Cai et al. [35] present an approach to NAS that aims to reduce both memory consumption and training time. In [36], the authors propose a search method that allows to search also for the depth of the architecture. Chang et al. [37] propose an approach that allows to convert binary flags to probabilities and vice-versa, so that the architecture can be optimized via back-propagation. In [38], the authors address the problem of the generalization in DARTS-like approaches [31] by dividing the problem into sub-problems. In [39], a continual-learning approach is proposed to avoid catastrophic forgetting in NAS. Other works conduct NAS by using simpler approaches, such as hill-climbing [40] or random search [41].

Finally, among the most widely-known NAS approaches, there is Evolutionary Computation.

The application of EC approaches to NNs is termed *neuroevolution*. Neuroevolution refers to the optimization, by using EC approaches, of the architecture and weights of a NN. NAS can be considered as a special case of neuroevolution where only the architecture is optimized by using EC approaches. A considerable number of approaches to evolutionary NAS have been proposed. In [42, 43], the authors use a mutation-only Genetic Algorithm (GA), using the DNNs as both genotype and phenotype. In [44, 45], special-purpose representations for DNNs are introduced. In [46], the NASNet encoding [25] is employed for evolutionary NAS. A similar attempt, although with limited computational resources, was done in [47]. A co-evolutionary approach to NAS has been recently introduced in [48]. In [49, 50, 51], the authors propose custom encodings to evolve "simple" (i.e., with single-input single-output layers) CNNs. In [52], the author introduces a GA that evolves cascades of convolutional filters for classification. In [53], the authors use a GA to tune the parameters of a machine learning model based on Boosted smoothing spline.

Other works have used Genetic Programming (GP) [54] to perform NAS. For instance, the approaches presented in [55, 56] use Strongly Typed GP [57] to evolve CNNs that perform either feature extraction or classification. In [58, 59, 60], the authors use Cartesian GP [61] to encode and evolve DNNs. The approaches proposed in [62], instead, make use of Grammatical Evolution [63] to evolve DNNs. In [64], GP is used to design CNNs to perform image saliency fusion.

As a final remark, as shown in [10, 11, 12], it is worth to note that evolutionary approaches to NAS yield NNs that have a good trade-off between performance and size.

3. Background

We present now the background concepts on CNN and LSTM (section 3.1), followed by the details of the multi-head CNN-LSTM architecture that is at the basis of this work (section 3.2).

3.1. CNN-LSTM

The idea of CNN used today were first established and applied by LeCun et al. [65], who presented a CNN and applied it to the classification task of two-dimensional (2D) images of handwritten digits. However, CNN models can be used also to analyze patterns in one-dimensional (1D) time series, as those ones available for RUL prediction purposes. Accordingly, we first describe 2D-CNN for extracting features from 2D image, then introduce the 1D-CNN employed in our study.

With reference to Figure 1, the first characteristic of a CNN is to use a *kernel* (i.e., a convolution matrix) to get the *kernel response* of the *local receptive field*. The latter is the part of the input on

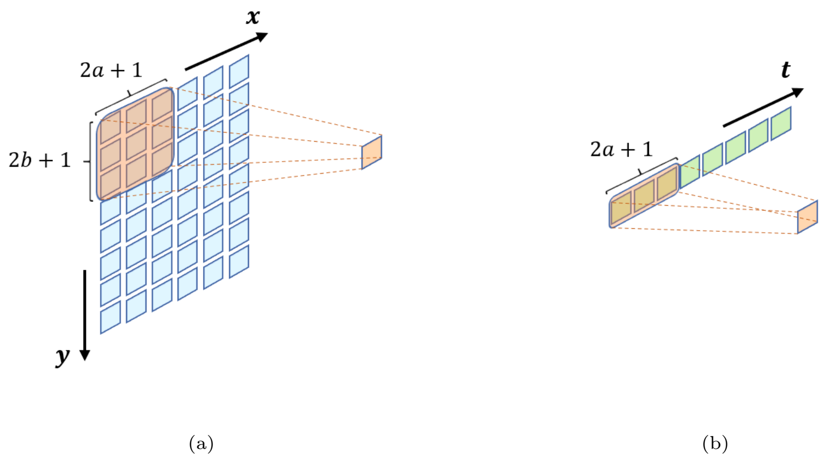


Figure 1. Illustration of the convolution operation for: (a) 2D-CNN; (b) 1D-CNN.

which the kernel is applied, and its response to the kernel is also called *feature map*. Figure 1 (a) visualizes the convolution operation of a 2D-CNN on an image. In the figure, an image I lies on a 2D plain, and a kernel K slides across the x and y axes. With a kernel size of $(2a + 1) \times (2b + 1)$, the feature map S is defined by:

$$S(x, y) = (I * K)(x, y) = \sum_{s=-a}^{a} \sum_{r=-b}^{b} I(x - s, y - r)K(s, r) \tag{1}$$

where $*$ denotes the convolution operation, and a and b are positive integers, such that the kernel has a fixed size (odd numbers).

In the 1D case, the same operation can be applied by simply ignoring the y axis. In Figure 1 (b), a 1D kernel K slides over a time series T. The algebraic expression of the convolution operation along t can then be written as:

$$S(t) = (T * K)(t) = \sum_{s=-a}^{a} T(t - s)K(s). \tag{2}$$

Instead of using fixed values for the kernel, the elements of the kernel (the so-called weights) can be learned. For instance, the kernel shown in Figure 1 (a) consists of $(2a+1) \times (2b+1)$ weights. When the output of the convolution defined by Eq. (1) or Eq. (2) is fed into a neuron in the following layer, the same weights w and bias b are used for all neurons. This second characteristic of CNNs is called *weight sharing*. Eventually, during the feedforward computation of a CNN the output at the spatial position (x, y) is defined by:

$$\sigma\left(\sum_{s}\sum_{r} I_{x-s,y-r} \cdot w_{s,r} + b\right) \tag{3}$$

where $\sigma(\cdot)$ denotes the sigmoid activation function. For the time series described in Figure 1 (b), the feedforward output obtained from the measurement at time t can be computed as:

$$\sigma\left(\sum_{s} T_{t-s} \cdot w_s + b\right). \tag{4}$$

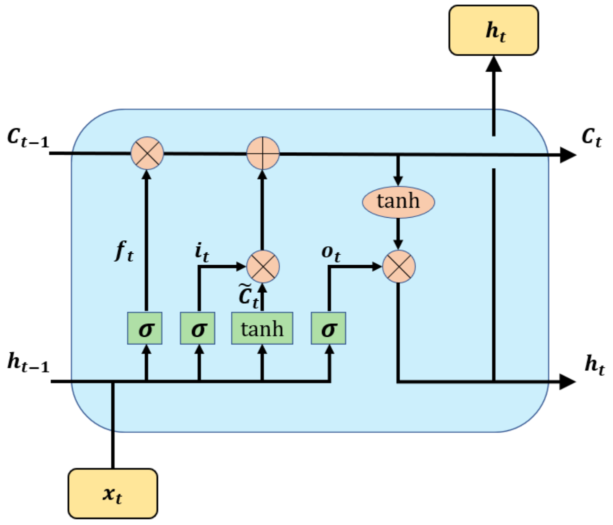


Figure 2. Scheme of an LSTM cell.

The second element of the DNN model we use in this work is the LSTM. As we will show later, we employ the LSTM to consider long term dependencies between concatenated features extracted by preceding CNNs. In fact, this is not possible with traditional RNNs, which suffer from an exponentially decaying gradient when those networks are trained with backpropagation through time. Because of this vanishing gradient problem, RNNs cannot keep track of long dependencies.

On the other hand, LSTM networks are able to keep long term dependencies between inputs in memory. This is made possible by the use of LSTM cells. Each LSTM cell, illustrated in Figure 2, can handle information by using a cell state, C_t, controlled by three gates: forget f_t, input i_t, and output o_t. More specifically, the LSTM cell works as follows. In the following, we denote with h_t and x_t the hidden state and the input observation respectively; the weight matrix and bias vector are denoted by W and b respectively (with a different subscript, depending on the gate); $\sigma(\cdot)$ denotes again the sigmoid activation function. Firstly, the forget gate decides what information will be removed from the cell state. This decision is made based on the concatenation (indicated with $[\cdot,\cdot]$) of the previous hidden state, h_{t-1}, and the current input observation, x_t:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f). \tag{5}$$

Similarly, the input gate decides to add information based on the concatenation of h_{t-1} and x_t:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{6}$$

A new candidate value $\widetilde{C_t}$ is then computed based on h_{t-1} and x_t, using the tanh activation function:

$$\widetilde{C_t} = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c). \tag{7}$$

Then, the LSTM cell updates its internal state, C_t, based on f_t, i_t and $\widetilde{C_t}$, as follows:

$$C_t = f_t \otimes C_{t-1} + i_t \otimes \widetilde{C_t} \tag{8}$$

where $\otimes$ indicates the element-wise multiplication operator. The motivation of using the tanh function is to normalize the cell state value in $[-1, 1]$, without the vanishing gradient problem. As shown in Eq. (5) and (6), the values of f_t and i_t range in $[0, 1]$, as a result of the sigmoid activation. Therefore, the first term of the addition in Eq. (8) determines how much information should be removed from C_{t-1} in C_t. In other words, the previous cell state C_{t-1} is completely "forgotten" when $f_t = 1$, while C_{t-1} is preserved without forgetting when $f_t = 1$. Similarly, the second term of the addition determines how much information calculated by Eq. (7) should be included in the current cell state C_t, based on the input gate value i_t.

Lastly, similarly to Eq. (5) and (6), the output of the LSTM cell is determined by the output gate, o_t, as:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{9}$$

and eventually the hidden state of the LSTM cell, h_t, is updated based on Eq. (9) and (8) as follows:

$$h_t = o_t \otimes \tanh(C_t). \tag{10}$$

where the value of the output gate, which ranges in $[0, 1]$ because of the sigmoid activation function, is multiplied by the tanh activation to normalize the hidden state value in $[-1, 1]$.

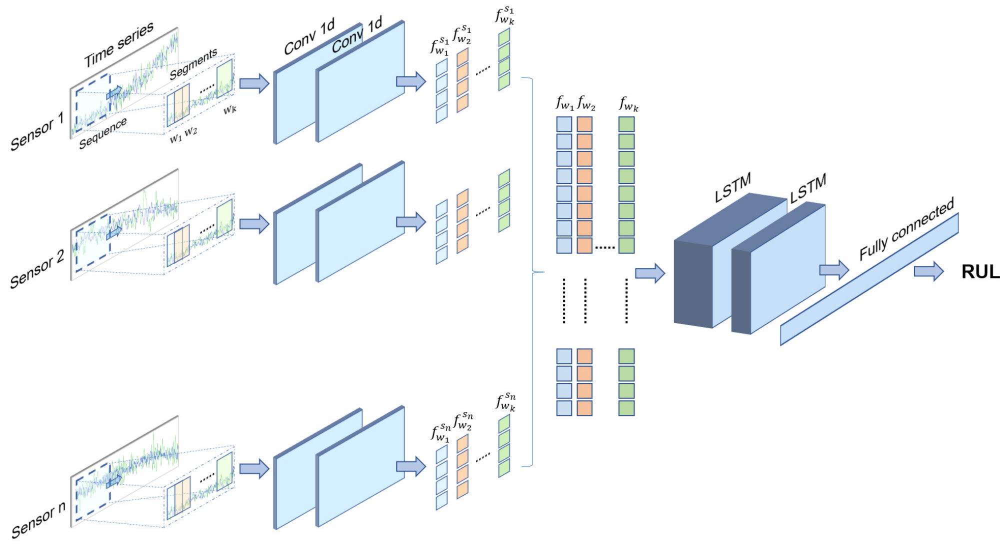

Time series
Segments
Sequence
Sensor 1
Sensor 2
Sensor n
w_1 w_2
w_k
Conv 1d
Conv 1d
$f_{w_1}^{s_1}$ $f_{w_2}^{s_1}$ $f_{w_k}^{s_1}$
f_{w_1} f_{w_2} f_{w_k}
$f_{w_1}^{s_n}$ $f_{w_2}^{s_n}$ $f_{w_k}^{s_n}$
LSTM
LSTM
Fully connected
RUL

Figure 3. The multi-head CNN-LSTM architecture.

3.2. Multi-head CNN-LSTM

In general, RUL prediction consists in developing a regression model for sensor data in multi-sensor environments. In this work, the time series data from sensor measurements indicate the state of the physical properties, such as temperature and pressure, of some target components monitored by sensors. In industrial applications, it is usually required to process each times series independently, because each sensor measures different properties.

As discussed in Section 1, DL-based models have been widely used to make accurate RUL predictions, and the combination of CNN and RNN have shown promising results. However, most of them do not discriminate the sensor readings from different sensors, because the high-level structure of those models is inspired by their application in other domains such as human activity recognition [66] or speech recognition [67]. To overcome this limitation, here we employ fully independent convolutional "heads", followed by a RNN. This architecture was proposed by Canizo et al. and dubbed as a multi-head CNN-RNN [9]. The key characteristic of the multi-head structure is that each head does not share its parameters with other heads, so that each convolutional layer has its own specialized filter to extract appropriate features. All the extracted features from all the heads are then concatenated before proceeding to the recurrent layers. Therefore, the layers following the CNNs remember the features of the past sequences and realize the temporal patterns throughout them.

As observed in [9], the multi-head CNN model is superior to a single multi-channel CNN consisting of multi-channel inputs and a single feature map applied to multiple time series. In fact, whereas the former keeps separate and independent extracted features for each time series, the latter mixes them all together into a single feature map and therefore loses the specialized features of each time series. The experiments in [9] proved that when this multi-head architecture is combined with a RNN, the resulting multi-head CNN-RNN provides better results compared to a multi-channel CNN-RNN. This shows that the independent feature maps used in the multi-head architecture are advantageous for extracting information from multiple time series.

As for the RNN part of the model, a variety of network types can be used for this purpose. In [9], the authors evaluated the results of five different types of recurrent layers in a multi-head CNN-RNN architecture. Based on their experimental results, here we use LSTM layers because they were shown to outperform other kinds of RNN when combined with the multi-head architecture.

Figure 3 shows the baseline multi-head CNN-LSTM architecture that we consider in this work. The figure shows that the parallel branches of the convolutional layers, the heads, are followed by LSTM layers. The convolutional heads are collectively called as a multi-head CNN. Each convolutional head is responsible for extracting 1D convolutional features from the time series of a specific sensor, which are then concatenated and fed to the LSTM layers. Lastly, the predicted RUL is obtained by the following fully connected layer with linear activation. To train the regression model, we consider as the loss function $l(\cdot)$ the Mean Squared Error (MSE) between the output of the model and the ground truth:

$$l(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{11}$$

where y indicates the ground truth (the actual RUL), $\hat{y}$ represents the predicted RUL, i.e., the output of the model, and n denotes the number of samples per batch of computation. For full-batch learning, n is the size of the whole set of training samples. Otherwise, the mean squared error can be calculated over a mini batch of n samples, with n smaller than the size of the training set.

One important aspect of the multi-head CNN-LSTM architecture is that it can be easily adapted to any multi-sensor environment with different kind of sensors. In our previous work [7], we have found in fact that the architectural replication of a convolutional head enables to use additional inputs and improve the RUL prediction without manipulating other hyper-parameters. Hence, even if different industrial applications might require to use a different number of sensors, one possibility could be to discover a well-performing architecture under a certain environment, and then reuse the optimized model in other environments without completely redesigning it, but simply by adding other convolutional heads (one for each additional sensor) on top of the discovered architecture.

On the other hand, as we will see in the next Section the performance of the multi-head CNN-LSTM architecture depends on various architecture parameters. Therefore, this kind of architecture represents a particularly interesting opportunity for applying NAS to PdM, and possibly also other kinds of industrial applications.

4. Evolutionary Neural Architecture Search for Predictive Maintenance (ENAS-PdM)

Since the CNN-LSTM architecture described above uses a serial combination of CNN heads and LSTM layers, the features extracted by the CNN heads affect the training of the following LSTM layers. Analyzing the relationship between the hyper-parameters of the CNN heads and those of the LSTM layers is thus especially difficult. Because of these relationships, and due to the size of the parameter space, manual engineering this kind of architecture is far from being trivial. In addition, this architecture uses a sliding window to process the input sequences, which makes trial-and-error architecture design even harder. This is because the performance of the model relies on the configuration of the sliding window size as well as the parameters of the CNN and LSTM parts.

Figure 4 illustrates how one head of the multi-head CNN processes the time series until the convolutional features are extracted. In this work, we refer to the fixed number of data points within the time series as a *sequence*. Each sequence is an input instance for one convolutional head and its length is denoted by l_s. For each sequence, a sliding window is then applied to slice the sequence into fixed-length *segments*. If a sliding window of length l_w runs over a sequence of length l_s, then the number of segments, k, is determined by $k = \frac{l_s - l_w}{stride} + 1$. Then, 1D convolutional filters are applied to each input in the convolutional layer. The number and length of the filters are denoted by m and l_f respectively. The convolutional layer is followed by batch normalization and activation layer, and these three layers are stacked C times. Therefore, the total number of filters in one head is $k \cdot m \cdot C$. The total number of parameters (i.e., the weights excluding the bias) to be trained in the multi-head CNN is then given by $p = l_f \cdot k \cdot m \cdot C \cdot n$, where n denotes the number of heads. Eventually, each convolutional head provides as output k convolutional features.

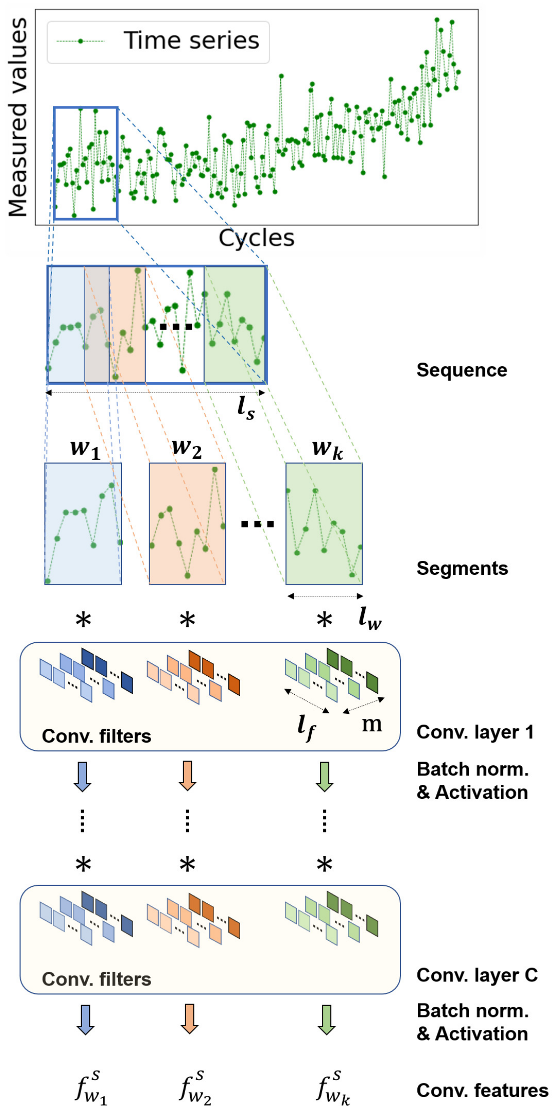


Figure 4. Framework of the multi-head CNN.

As shown in Figure 3, the extracted features of each k-th segment in each of the n heads are then concatenated. Before the convolutions, we add zeros at the beginning and end of the segment, and set the *stride* of the filter to one, so that each filter returns the extracted feature of length l_w equal to the length of the segment. With reference to Figure 3, $f_{w_1}, f_{w_2}, \dots f_{w_k}$ denote the concatenated features, each of them of length determined by: $l_c = l_w \cdot m \cdot n$.

The following stacked LSTM layers then process the sequence of those concatenated features. The LSTM layers handle long term dependencies between the k concatenated features to make a prediction based on the past information. The number of hidden units in the LSTM is a hyper-parameter that affects the ability to keep track of these long term dependencies. To consider different levels of abstraction, we use two stacked LSTM layers with $L_1 \cdot n_{lstm}$ and $L_2 \cdot n_{lstm}$ hidden units respectively, where n_{lstm} is a fixed multiplicand.

As described above, the parameters of the sliding window, CNN and LSTM are related to each other. More specifically, the window length l_w determines the number of segments k, which is also the span of the data for the LSTM. The feature length l_w also affects the decision of the hyper-parameters of the CNN, because different numbers and lengths of convolutional filters are required to extract proper features according to the input size. Overall, the parameters of the CNN and LSTM eventually affect the performance of the model in a complex way. However, examining manually a large number of possible architectures in order to obtain a meaningful empirical evidence on the parameter effect, or trying to model the complex dependencies between these parameters, is difficult and time-consuming. In this context, we propose the ENAS-PdM algorithm to overcome these limitations.

4.1. Individual encoding

Given the baseline structure of the model proposed above, our goal is to find optimized that are able to make better RUL predictions. To do that, we consider the optimization of the following architecture parameters:

- l_w, length of sliding window;
- l_f, length of convolutional filters;
- m, number of convolutional filters;
- C, number of convolutional layers;
- L_1, multiplier of hidden units number (1st LSTM layer);
- L_2, multiplier of hidden units number (2nd LSTM layer).

In preliminary experiments, we observed that the best architectures are found with $l_w = l_f$. Therefore, we fix this constraint such that the number of architecture parameters to be optimized is reduced to 5. As for L_1 and L_2, the empirical evidence collected in our previous work [7] tells us that underfitting occurs when the number of LSTM hidden units is less than 80, while the architectures with more than 400 hidden units are prone to overfitting. Because the range from 80 to 400 for the two LSTM layers is too large to be explored, we divide it by 20 to decrease the number of possible genotype representations from the above range. The multiplier is then assigned to the genotype, and the multiplicand, 20, is used to translate the genotype to phenotype. The bounds for the other parameters are also set based on our previous knowledge. The lower and upper bounds for each parameter considered in our experiments are shown in Table 1.

Table 1. Bounds of the parameters optimized by ENAS-PdM.

Parameter	Min	Max
l_w	1	5
m	1	10
C	1	2
L_1	4	20
L_2	4	15

From an evolutionary NAS perspective, in our case the genotype is then a list of integers within the bounds specified in Table 1. The phenotype is the resulting architecture. When translating genotypes to phenotypes, we apply a correction mechanism on L_2, to make sure that it is equal or smaller than L_1. This is because in our model we stack the layers to get hierarchical features: the second layer is in fact used to derive high level abstraction from the feature representation of the first layer. If L_2 is greater than L_1, this hierarchical feature abstraction is not possible, making the model prone to overfitting. It this happens, we correct L_2 according to the following saturation scheme:

$$L_2 = \begin{cases} L_2, & L_2 < L_1 \\ L_1, & L_2 \geq L_1 \end{cases}$$

4.2. Proposed algorithm

The pseudo-code of the proposed ENAS-PdM algorithm is shown in Algorithm 1. In the following we describe the details of each element indicated in the pseudo-code.

Algorithm 1 Pseudo-code of the proposed ENAS-PdM algorithm.

```
function Evolution($a, b$)
    $pop \leftarrow initialize\_pop()$
    $evaluated \leftarrow Set()$                                        ▷ Evaluated individuals
    for ($gen = 0;\ gen < generations;\ gen+ = 1$) do
        $evaluate\_fitness(evaluated, pop)$
        $new\_pop \leftarrow select(pop)$
        $new\_pop \leftarrow crossover(new\_pop)$
        $new\_pop \leftarrow mutation(new\_pop)$
        $pop \leftarrow check\_parents(pop, new\_pop)$
    end for

    return $best(pop)$
end function

procedure evaluate_fitness($evaluated, pop$)
    for $ind \in pop$ do
        if $ind \notin evaluated$ then
            $ind.fitness \leftarrow fitness(ind)$
            $evaluated.add(ind)$
        else
            $ind.fitness \leftarrow evaluated.get(ind).fitness$
        end if
    end for
end procedure
```

4.2.1. Initialization

We initialize the population by generating $n_{pop}-1$ individuals at random. In our experiments, we set n_{pop} to 50. The remaining individual is initialized by taking the parameters of the architecture proposed in [7], i.e., we use the so-called super-fit mechanism [68] in order to start the evolutionary process with a good individual in the initial population. This initialization strategy is implemented in the *initialize_pop()* function that is used in line 5 of Algorithm 1.

4.2.2. Fitness evaluation

The fitness of an individual is computed by constructing the phenotype (a multi-head CNN-LSTM architecture) associated to the given genotype (a vector containing the architecture's parameters) and evaluating it (see Section 5.2 for details). To reduce the computational costs of the evolutionary search, we implement a history mechanism in order to avoid the redundant computation of the fitness of individuals previously evaluated. The pseudo-code of this function is shown in line 15 of Algorithm 1.

4.2.3. Genetic operators

In order to have a good trade-off between exploration and exploitation, we use both crossover and mutation. Each operator is applied independently with a probability of $p_{cx} = p_{mut} = 0.5$. The probabilities have been chosen such that, usually, individuals are produced by either mutation or crossover (exclusively). This allows us to avoid disruptive combinations of mutation and crossover that could lead to bad individuals.

As for crossover (line 7 of Algorithm 1), we use a specialized one-point crossover that first ranks the individuals by their fitness (line 6), and then mates, according to the crossover probability, the individual in the ($2i$)-th position with the one in the ($2i + 1$)-th position, with $i \in [0, \frac{n_{pop}}{2} - 1]$. This means that crossover can occur between the best individual and the second best (0-th position and 1st position, respectively), the third best and the fourth best (2nd position and 3rd position, respectively) and so on. This operator allows us to *exploit* the best individuals, trying to combine them in even better individuals, and to *explore*, by combining bad solutions which can lead to regions of the state space that are far from the region in which the current best individuals lie.

We then apply uniform mutation on the population (containing the offspring generated by crossover and individuals that did not undergo crossover), in which each gene (i.e., one of the architecture parameters), according to a probability p_{gene}, can be mutated to another value uniformly drawn from its bounds (line 8 of Algorithm 1). The p_{gene} parameter has been set to 0.3, so that the expected number of mutations is set between 1 and 2. The rationale behind this choice is the following: we need more than one expected mutated gene, to have individuals that are different from their parents (in case they are produced only by means of mutation). Moreover, mutating one gene at a time may lead to a slow search process, which should be avoided since the fitness evaluation phase is quite computationally heavy. For this reason, we set p_{gene} such that, on average, we have 1.5 mutated genes (out of 5) per individual ($\mathbb{E}[\sum_{i=1}^{5} rand() < p_{gene}] = 1.5$). This allows us to have a relatively faster search process while avoiding disruptive mutations in the individuals.

When creating the population for the next generation, we check the fitness of the parents for each offspring. If the offspring has better fitness than one of its parents, we replace the worst parent with it. This way, we ensure that the mean fitness of the population is monotonically decreasing (i.e., we use implicit elitism). This mechanism is implemented in the *check_parents()* function (line 9 of Algorithm 1).

4.2.4. Stop criterion

We use a stop criterion on the number of generations, in our experiments set to 50. When this criterion is met, the algorithm returns the best individual, according to the specific fitness, found during the evolutionary process.

5. Experimental setup

In the following, we present the details of our experimentation: the C-MAPSS benchmark dataset (section 5.1), the evaluation metrics used as fitness for the NAS process (section 5.2), the details of the training process (section 5.3), and the computational setup with some considerations on reproducibility (section 5.4).

5.1. C-MAPSS benchmark dataset

In our experiments, we use the Commercial Modular Aero-Propulsion System Simulation (C-MAPSS) [69] as the benchmark dataset. The C-MAPSS contains the simulation of various NASA turbofan engine degradation, and it is one of the most widely used dataset in RUL prediction research. This allows us to validate our proposed method in comparison with the state-of-the-art algorithms from the literature. As described in Table 2, the dataset includes four sub-datasets, FD001, FD002, FD003 and FD004, according to different simulation settings.

While the training set of each sub-dataset consists of run-to-failure histories of different engines, the simulation of each test engine is terminated before its failure because the RUL of each engine in the test set is required to be predicted for reporting the final results. The data of each engine consists of 21 multi-variate time series collected from multi-sensor environment. Among them, 7 time series that do not show changes over time are discarded, thus we use only 14 time series as inputs. All the sensor readings and the RUL prediction are updated at the same frequency. In the next, we refer to a time unit of both the RUL prediction and sensor measurements as *cycle*.

Table 2. C-MAPSS dataset overview [7].

Sub-dataset	FD001	FD002	FD003	FD004
Number of engines in training set	100	260	100	249
Number of engines in test set	100	259	100	248
Max/min cycles in training set	362/128	378/128	525/145	543/128
Max/min cycles in test set	303/31	367/21	475/38	486/19
Operating conditions	1	6	1	6
Fault modes	1	1	2	2

Based on [70], we apply a piece-wise linear function to the target RUL. This set the all the target RUL greater than 125 to 125. Then, the target RUL linearly decrease over cycles. For most engines, the target RUL does not decrease at the very beginning of the time series, because the degradation of the engine starts after a number of operations. This approach is commonly used in the recent literature [5, 6, 7].

Lastly, the minimum cycles in the test set listed in the Table 2 indicate the minimum cycles among all engines in each sub-dataset, a value related to the sequence length l_s introduced before. More specifically, for each sub-dataset l_s must be smaller than the minimum cycles, so that the determined architecture with input length l_s can be used for all the engines in the test set. Hence, we set the l_s as 31, 21, 38 and 19 respectively for FD001, FD002, FD003 and FD004.

5.2. Evaluation metrics

The goal of our work is to search for the CNN-LSTM architecture that can predict the RUL most accurately. To evaluate such a model, we use two evaluation metrics based on the error between the predicted and target RUL, which is defined by $d_i = RUL_i^{predicted} - RUL_i^{target}$, where d_i denotes the difference between two RUL values of the i-th instance.

The first metric is Root-Mean-Square Error (RMSE), which is given by:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N} d_i^2} \tag{12}$$

where N is the total number of test samples fed into the model during the test. The other metric is the so-called *Score* function [69]. This metric was proposed to differentiate between early and late predictions, and it is computed as follows:

$$Score = \sum_{i=1}^{N} SF_i, \quad SF = \begin{cases} e^{-\frac{d_i}{13}} - 1, & d_i < 0 \\ e^{\frac{d_i}{10}} - 1, & d_i \geq 0 \end{cases} \tag{13}$$

i.e., it assigns a larger Score value to late predictions w.r.t. early ones, based on the intuition that late predictions are usually disadvantageous in terms of preventing failures.

These two metrics are used to evaluate the performance of the trained model, therefore we employ both of them for the fitness evaluation in ENAS-PdM. To search for the best architecture by using only the data in the original C-MAPSS training set, we divide the training set into two sets, respectively for training and validation. More specifically, 90% of the engines in the original training set are used to train the model constructed from each individual genotype. The remaining 10% of the engines are used to evaluate either validation RMSE or Score. Thus, 10, 26, 10 and 25 engines are separated from the original training set as the validation set for evaluating the fitness.

5.3. Training details

The architectures based on the individuals generated during the evolutionary process are trained at the fitness evaluation stage of ENAS-PdM. During the training process, we conduct a supervised learning with labeled training samples. For training the model to predict the RUL, we employ the MSE, see Eq. (11), as the loss function. The weights of the model are then optimized to minimize the loss using RMSprop algorithm. Based on the empirical evidences from our previous work [7], we set the batch size of gradient descent to 400, and we limit the maximum number of training epochs to 20.

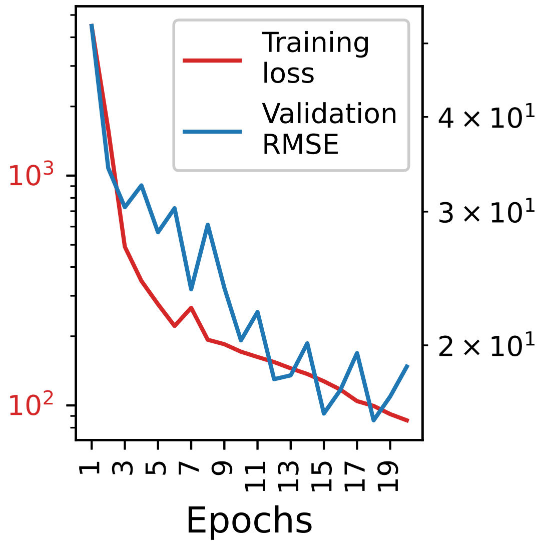
In order to reduce the computing time, we introduce an early stopping mechanism in the optimizer based on learning rate decay. In detail, we drop the learning rate down to 10^{-4} after 10 epochs, and then we divide it again by 10 after 5 epochs. Figure 5 shows the effect of learning rate decay: both figures show that the optimizer requires a certain amount of patience, 5 epochs in the example, to ignore the sudden spikes in the validation. Otherwise, the training stops too early, e.g., at epoch 4. In the Figure 5 (a), the training loss cannot converge because of overfitting, and the validation RMSE keeps fluctuating for all the epochs. The model cannot stop training within 20 epochs, because it keeps finding lower validation RMSE values within the patience. This means that the early stopping would not work well even if it was applied. On the other hand, when we apply the learning rate decay, as depicted in Figure 5 (b), the validation RMSE reaches the lowest value at epoch 14, and the training is early stopped at epoch 19. The training can be stopped within the patience after the training loss has converged. Because we schedule the learning rate properly, the validation RMSE does not fluctuate anymore and is flattened (or, it increases) after overfitting starts.

5.4. Computational setup and reproducibility

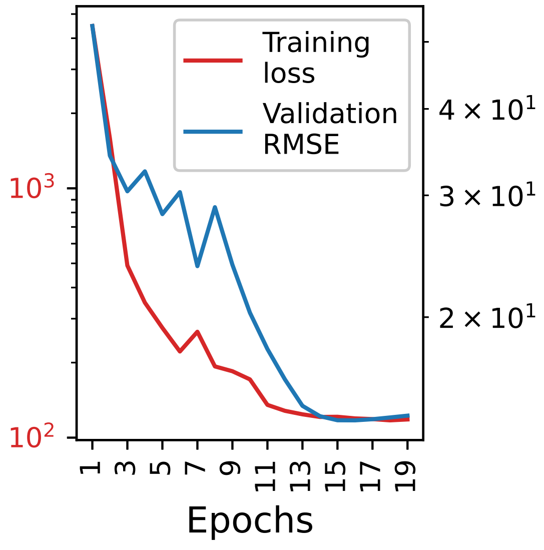
All the experiments have been conducted on a workstation with Intel(R) Core i9-7940X CPU and NVIDIA TITAN Xp GPU. The multi-head CNN-LSTM models are implemented in TensorFlow 2.3. The ENAS-PdM algorithm is implemented in Python, using the DEAP library [71]. The source code of our implementation is publicly available on GitHub[1].

An important aspect to highlight here is that, while most NAS tasks require large-scale computational resources, the proposed ENAS-PdM algorithm can conduct the search effectively within

[1]`https://github.com/mohyunho/ENAS-PdM`



(a)



(b)

Figure 5. The loss on the training data and RMSE on the validation data: (a) the optimizer without learning rate decay; (b) the optimizer with learning rate decay.

a reasonable time (from 8 to 18 hours, depending on the sub-dataset) by the single GPU above. The reason for this is twofold. Firstly, the history mechanism discussed in Section 4.2.2 allows to evaluate a smaller number of individuals w.r.t. the size of the population in each generation, by simply re-using, in case of duplicates, the fitness of the individuals evaluated in the previous generations. Secondly, the early stopping policy introduced in Section 5.3, by avoiding overfitting, enables to use a large initial learning rate of 10^{-3} that in turn allows to save a few training epochs for each individual, and a considerable amount of time overall.

Another note regards reproducibility. In general, reproducible results should be guaranteed and might even be needed in some industrial contexts. Nevertheless, some operations of the DNNs implemented by the DL TensorFlow framework result in non-deterministic outputs when executed on a GPU. This issue is caused by the non-deterministic order of the operations running in parallel on the GPU, in addition to the limited-precision floats. To get reproducible results, we considered using the determinism library that provides deterministic outputs by addressing the issues above. However, we noted that the determinism library dramatically slows down the GPU computation. Table 3 presents the lower and upper bounds of the training time on the largest sub-dataset, FD004. For the smallest architecture, constructed from the lower bounds shown in Table 1, the use of determinism does not affects too much the training time. However, it almost doubles the training time of the largest architecture (constructed from the upper bounds). Because our goal here is to search for the best architecture, we deemed not necessary to use determinism. Hence, instead of enduring the slow down, we repeat the ENAS-PdM process without determinism three times for each sub-dataset and fitness function, and compare its results across the different trials.

Table 3. Training time of the architectures on FD004.

Framework	Training time (s)	
	Smallest architecture in search space	Largest architecture in search space
TensorFlow	75	183
TensorFlow + Determinism[2]	96	373

6. Experimental results

The first goal of our experiments is to test if an architecture optimized for one sub-dataset is actually able to generalize to other sub-datasets. In other words, we verify if the model found by ENAS-PdM on a certain sub-dataset also provides promising results on the others. Furthermore, we verify if (and how) the choice of the fitness function between the two different metrics defined above, validation RMSE and Score, affects the test results. Finally, we compare the performance of the best architectures discovered by ENAS-PdM with that of the methods from the state-of-the-art.

To achieve the first goal, we test the architecture discovered by ENAS-PdM on all the sub-datasets, regardless of which sub-dataset was used for the optimization conducted by ENAS-PdM. Due to the non-deterministic GPU operations discussed in Section 5.4, we run ENAS-PdM three times under exactly the same settings, to show the reliability of the optimization process. Furthermore, we run separate evolutionary processes using either the validation RMSE or Score as fitness function. Altogether, we run ENAS-PdM six times, i.e., three trials for each of the two fitness functions, for each sub-dataset. Eventually, we collect the best architectures discovered by ENAS-PdM on the four sub-datasets at the end of the 24 total trials. Among them, the two architectures that

[2] `https://github.com/NVIDIA/framework-determinism`

Table 4. Results of the best architectures found by ENAS-PdM with validation RMSE on FD001.

EA specifications	Pop. × Gen.	50 × 50									
	Fitness	Validation RMSE on FD001									
Test specifications	Metrics	RMSE					Score				
	Sub-datasets	FD001	FD002	FD003	FD004	Sum	FD001	FD002	FD003	FD004	Sum
Test results	State-of-the-art [5], [6]	11.94	19.29	**12.10**	22.14	65.47	**220**	2250	251	**2840**	**5561**
	A_1 (ENAS-PdM trial-1)	**11.48**	**17.47**	12.48	20.59	**62.02**	240	2074	412	3592	6318
	A_2 (ENAS-PdM trial-2)	11.54	17.79	**12.10**	20.93	62.36	250	2260	**234**	3908	6652
	A_3 (ENAS-PdM trial-3)	11.71	17.55	12.88	**20.38**	62.52	254	**1695**	564	3395	5908

Table 5. Results of the best architectures found by ENAS-PdM with validation Score on FD001.

EA specifications	Pop. × Gen.	50 × 50									
	Fitness	Validation Score on FD001									
Test specifications	Metrics	RMSE					Score				
	Sub-datasets	FD001	FD002	FD003	FD004	Sum	FD001	FD002	FD003	FD004	Sum
Test results	State-of-the-art [5], [6]	11.94	19.29	**12.10**	22.14	65.47	220	**2250**	**251**	**2840**	**5561**
	A_4 (ENAS-PdM trial-1)	11.28	17.83	12.24	20.55	**61.90**	230	2556	325	3778	6889
	A_5 (ENAS-PdM trial-2)	11.70	**17.68**	13.81	**20.48**	63.67	242	2324	834	3883	7283
	A_6 (ENAS-PdM trial-3)	**11.10**	19.16	13.65	20.67	64.58	**214**	3013	910	3634	7771

Table 6. Results of the best architectures found by ENAS-PdM with validation RMSE on FD002.

EA specifications	Pop. × Gen.	50 × 50									
	Fitness	Validation RMSE on FD002									
Test specifications	Metrics	RMSE					Score				
	Sub-datasets	FD001	FD002	FD003	FD004	Sum	FD001	FD002	FD003	FD004	Sum
Test results	State-of-the-art [5], [6]	**11.94**	19.29	12.10	22.14	65.47	**220**	**2250**	251	**2840**	**5561**
	A_7 (ENAS-PdM trial-1)	12.01	**17.76**	13.24	**19.73**	**62.74**	263	5158	926	3004	9351
	A_8 (ENAS-PdM trial-2)	12.55	18.20	12.81	20.46	64.02	292	5740	425	4027	10484
	A_9 (ENAS-PdM trial-3)	12.57	18.48	**11.68**	21.40	64.13	302	7124	**241**	5163	12830

Table 7. Results of the best architectures found by ENAS-PdM with validation Score on FD002.

EA specifications	Pop. × Gen.	50 × 50									
	Fitness	Validation Score on FD002									
Test specifications	Metrics	RMSE					Score				
	Sub-datasets	FD001	FD002	FD003	FD004	Sum	FD001	FD002	FD003	FD004	Sum
Test results	State-of-the-art [5], [6]	11.94	19.29	**12.10**	22.14	65.47	**220**	2250	**251**	**2840**	**5561**
	A_{10} (ENAS-PdM trial-1)	11.65	18.08	13.40	**20.02**	63.15	238	1946	998	4346	7528
	A_{11} (ENAS-PdM trial-2)	**11.56**	**17.67**	12.98	20.19	**62.40**	247	1745	808	3051	5851
	A_{12} (ENAS-PdM trial-3)	11.80	17.76	13.42	20.27	63.25	258	**1683**	929	3459	6329

Table 8. Results of the best architectures found by ENAS-PdM with validation RMSE on FD003.

EA specifications	Pop. × Gen.	50 × 50									
	Fitness	Validation RMSE on FD003									
Test specifications	Metrics	RMSE					Score				
	Sub-datasets	FD001	FD002	FD003	FD004	Sum	FD001	FD002	FD003	FD004	Sum
Test results	State-of-the-art [5], [6]	11.94	19.29	**12.10**	22.14	65.47	**220**	2250	**251**	**2840**	**5561**
	A_{13} (ENAS-PdM trial-1)	12.16	**17.83**	12.82	**20.53**	63.34	280	**1766**	498	4062	6606
	A_{14} (ENAS-PdM trial-2)	13.06	18.85	14.11	20.78	66.80	295	6610	1053	4085	12043
	A_{15} (ENAS-PdM trial-3)	**11.76**	17.84	12.64	20.58	**62.82**	271	2641	408	3613	6933

Table 9. Results of the best architectures found by ENAS-PdM with validation Score on FD003.

EA specifications	Pop. × Gen.	50 × 50									
	Fitness	Validation Score on FD003									
Test specifications	Metrics	RMSE					Score				
	Sub-datasets	FD001	FD002	FD003	FD004	Sum	FD001	FD002	FD003	FD004	Sum
Test results	State-of-the-art [5], [6]	11.94	19.29	**12.10**	22.14	65.47	**220**	2250	**251**	**2840**	**5561**
	A_{16} (ENAS-PdM trial-1)	12.40	19.25	12.87	21.30	65.82	262	2381	264	5504	8411
	A_{17} (ENAS-PdM trial-2)	12.32	**17.97**	12.31	20.66	**63.26**	287	**1759**	290	4169	6505
	A_{18} (ENAS-PdM trial-3)	**11.49**	18.91	13.74	**20.44**	64.58	234	2232	761	4316	7543

Table 10. Results of the best architectures found by ENAS-PdM with validation RMSE on FD004.

EA specifications	Pop. × Gen.	50 × 50									
	Fitness	Validation RMSE on FD004									
Test specifications	Metrics	RMSE					Score				
	Sub-datasets	FD001	FD002	FD003	FD004	Sum	FD001	FD002	FD003	FD004	Sum
Test results	State-of-the-art [5], [6]	11.94	19.29	**12.10**	22.14	65.47	**220**	**2250**	**251**	2840	**5561**
	A_{19} (ENAS-PdM trial-1)	**11.39**	**17.69**	13.74	20.02	**62.84**	224	5170	1028	2897	9319
	A_{20} (ENAS-PdM trial-2)	12.49	17.70	13.76	**18.97**	62.92	246	4052	834	**2712**	7844
	A_{21} (ENAS-PdM trial-3)	**11.39**	**17.69**	13.74	20.02	**62.84**	224	5170	1028	2897	9319

Table 11. Results of the best architectures found by ENAS-PdM with validation Score on FD004.

EA specifications	Pop. × Gen.	50 × 50									
	Fitness	Validation Score on FD004									
Test specifications	Metrics	RMSE					Score				
	Sub-datasets	FD001	FD002	FD003	FD004	Sum	FD001	FD002	FD003	FD004	Sum
Test results	State-of-the-art [5], [6]	11.94	19.29	**12.10**	22.14	65.47	220	2250	**251**	2840	**5561**
	A_{22} (ENAS-PdM trial-1)	11.32	17.93	13.73	19.73	62.71	210	5360	967	**2391**	8928
	A_{23} (ENAS-PdM trial-2)	12.02	**17.66**	13.39	**19.27**	**62.34**	242	4369	930	2595	8136
	A_{24} (ENAS-PdM trial-3)	**11.05**	18.06	13.32	22.26	64.69	**193**	**1897**	532	3920	6542

show the best test results in terms of either the sum of RMSE or the sum of Score are eventually considered as the results of our proposed method and compared against the state-of-the-art.

Regarding the parametrization of the EA, as discussed earlier for all the experiments we set both the population size and the number of generations to 50, which allows enough evaluations to ensure the convergence of the fitness across generations. Using the same fixed values for all the experiments also allows us to compare the results fairly by running each ENAS-PdM under the same evolutionary process.

Table 4 provides the test results of the best architecture found by every independent trial of ENAS-PdM on FD001 using the validation RMSE as fitness function. In the next, A denotes the architecture derived in the independent trials, and its subscript numbering follows the order of the trials in our experiments. For reference, the state-of-the-art results shown in the tables are taken from two recent papers, [5] and [6]: more specifically, for each sub-dataset we combine the best results from these two works, and include them into the tables to compare them with our results. The last three rows in Table 4 show the results of the three trials of ENAS-PdM. Due to the non-deterministic computing on the GPU, we indeed get three different architectures, denoted by A_1, A_2 and A_3. Although the architectures are different, their performance are close to each other, and the test results are significantly better than the current state-of-the-art in terms of sum of RMSE. On the other hand, their Score results are promising, but not as much as the RMSE, since they do not provide a lower Score w.r.t. the current state-of-the-art sum of Score.

Since the state-of-the-art methods are not easily outperformed by ENAS-PdM when using the validation RMSE as fitness function, we replicate the same experiments using the validation Score as fitness function, instead of the validation RMSE. As shown in Table 5, however, the results show that the optimization in terms of validation Score performs slightly worse than the one conducted on the validation RMSE. This difference might be due to the fact that since the RMSE is a quadratic function (thus unimodal), the fitness landscape is easier to minimize w.r.t. the landscape produced by the Score function, which may contain local minima.

For each run of ENAS-PdM performed in this work, we report in the Appendix the related boxplots of the fitness distribution across generations, using either RMSE or Score as the fitness. For instance, Figure A.7 and A.10 result in A_1 and A_4 respectively. In both figures, we can see that the populations adapt quite quickly to the problem. In fact, we can see that in both cases the standard deviation of the fitness becomes small in the early generations, while in the last generations the mean gets closer the best individual found. This suggests that the diversity of the individuals may be low, indicating that the evolution has come to a stagnation phase.

Table 12. Average number of evaluations across trials.

	FD001	FD002	FD003	FD004
Number of evaluations (average across 3 trials)	869/2500	830/2500	837/2500	656/2500

Table 13. Specifications of the architectures discovered by ENAS-PdM and their comparison.

Architecture	Used sub-dataset for EA	Used fitness for EA	Phenotype (l_w, m, l_f, C, $L_1 \cdot 20$, $L_2 \cdot 20$)	RMSE (sum)	Score (sum)
A_1	FD001	Validation RMSE	(2, 3, 2, 1, 400, 260)	62.02	6318
A_2			(2, 3, 2, 1, 400, 240)	62.36	6652
A_3			(2, 5, 2, 1, 300, 240)	62.52	5908
A_4		Validation Score	(2, 3, 2, 1, 340, 140)	**61.90**	6889
A_5			(2, 3, 2, 1, 160, 140)	63.67	7283
A_6			(2, 3, 2, 1, 220, 80)	64.85	7771
A_7	FD002	Validation RMSE	(1, 9, 1, 1, 160, 140)	62.74	9351
A_8			(1, 8, 1, 1, 260, 260)	64.02	10484
A_9			(1, 7, 1, 1, 320, 260)	64.13	12830
A_{10}		Validation Score	(2, 4, 2, 1, 400, 120)	63.15	7528
A_{11}			(2, 5, 2, 1, 360, 260)	62.40	**5851**
A_{12}			(2, 5, 2, 1, 320, 280)	63.25	6329
A_{13}	FD003	Validation RMSE	(2, 4, 2, 1, 260, 240)	63.34	6606
A_{14}			(2, 3, 2, 2, 280, 240)	66.80	12043
A_{15}			(2, 3, 2, 1, 360, 300)	62.82	6933
A_{16}		Validation Score	(1, 3, 1, 1, 200, 200)	65.82	8411
A_{17}			(2, 4, 2, 1, 280, 280)	63.26	6505
A_{18}			(1, 4, 1, 1, 260, 120)	64.58	7543
A_{19}	FD004	Validation RMSE	(2, 10, 2, 2, 100, 100)	62.84	9319
A_{20}			(1, 9, 1, 2, 80, 80)	62.92	7844
A_{21}			(2, 10, 2, 2, 100, 100)	62.84	9319
A_{22}		Validation Score	(2, 10, 2, 2, 160, 80)	62.71	8928
A_{23}			(1, 7, 1, 2, 280, 80)	62.34	8136
A_{24}			(2, 5, 2, 1, 340, 80)	64.69	6542

We conduct similar experiments also for FD002, FD003 and FD004. The results of these experiments are reported in Table 6-11. Regardless of the sub-dataset used for the EA, we can achieve a lower RMSE compared to the state-of-the-art, whereas we always get a higher value for the Score metric. In addition, in each experiment at least one of the three discovered architectures always largely outperforms the existing methods in terms of RMSE, except for FD003. Despite this one exception, we observe successful results in the aggregated RMSE across all sub-datasets, due to the significant improvement on 3 out of 4 sub-datasets. Moreover, the results obtained from the experiments suggest that the proposed method can easily generalize from one sub-dataset to the others.

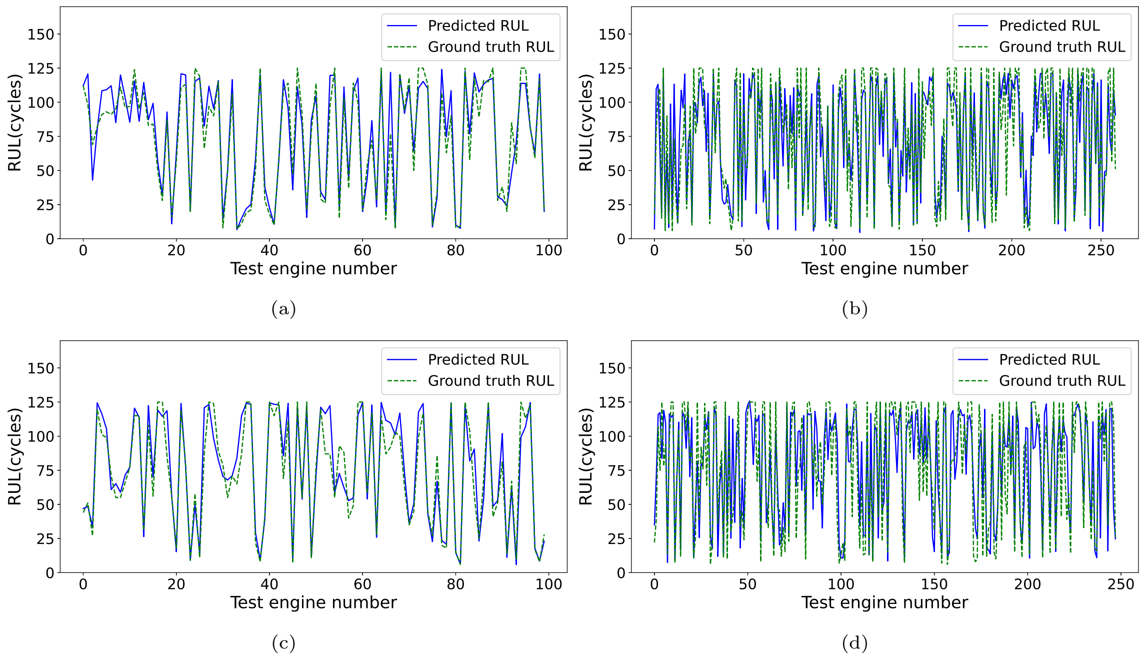


Figure 6. Predicted RUL of each engine in the last given cycle by the proposed method, A_4: (a) FD001 dataset; (b) FD002 dataset; (c) FD003 dataset; (d) FD004 dataset.

For instance, Table 10 proves that the best architectures for FD004 also show a good performance on the other sub-datasets.

When we use the validation RMSE as fitness function, we also count the number of evaluations across the trials, to measure the saved computational costs of ENAS-PdM. As it can be seen in Table 12, while a total of 2500 individuals (50 individuals $\times$ 50 generations) should be evaluated during the evolutionary search, we only compute less than 900 individuals on average, by saving and reusing the fitness. Therefore, the number of fitness evaluations is less than 40% of the total number of individuals appeared in the evolutionary process. Moreover, we can observe that our approaches finds good solutions while evaluating less than 4.3% of the 20400 possible combinations described in Table 1. While a search space of this size may seem small to justify the use of a NAS approach, it should be noted that an exhaustive search would require up to 400 hours of computing time, as opposed to a maximum of 18 hours for one trial of our proposed method.

Table 13 gives a comparison of all the architectures discovered by all the experiments above, along with the phenotype corresponding to the configuration of each architecture. Among them,

A_4 and A_{11} provide the best performance in terms of the sum of RMSE and Score respectively. The detailed results of A_4 are reported in Table 5, while the results of A_{11} are reported in Table 7.

The RUL predictions of A_4 on the four sub-datasets are depicted in Fig. 6, to visualize the results. The horizontal axis indicates the number of test engines listed in Table 2, while the vertical axis represents the RUL. In each subfigure, the difference between the two graphs represents the error on the RUL prediction.

Additionally, the best experimental results of our proposed method are compared to the results of the best methods from the current state-of-the-art. Table 14 and 15 display the compared methods and their results in terms of the test RMSE and Score respectively. The methods in the tables are reported in chronological order, and they gradually provide better RMSE and Score over year. To reflect the robustness of the methods across the sub-datasets, we also report the sum of RMSE and Score across the C-MAPSS sub-datasets.

Table 14. RUL prediction comparison with state-of-the-art methods (sorted by year), in terms of RMSE.

Method	RMSE				
	FD001	FD002	FD003	FD004	Sum
CNN, 2016 [3]	18.45	30.29	19.82	29.16	97.72
LSTM, 2017 [4]	16.14	24.49	16.18	28.17	84.98
BiLSTM, 2018 [72]	13.65	23.18	13.74	24.86	75.43
DCNN, 2018 [73]	12.61	22.36	12.64	23.31	70.92
Semi-supervised DL, 2019 [6]	12.56	22.73	**12.10**	22.66	70.05
DAG network, 2019 [8]	11.96	20.34	12.46	22.43	67.09
Multi-head CNN-LSTM, 2020 [7]	13.27	19.49	13.21	23.89	69.86
AdaBN-DCNN, 2020 [5]	11.94	19.29	12.31	22.14	65.68
The proposed method (A_4)	**11.28**	17.83	12.24	20.55	**61.90**
The proposed method (A_{11})	11.56	**17.67**	12.98	**20.19**	62.40

Table 15. RUL prediction comparison with state-of-the-art methods (sorted by year), in terms of Score.

Methods	Score				
	FD001	FD002	FD003	FD004	Sum
CNN, 2016 [3]	1290	13600	1600	7890	24380
LSTM, 2017 [4]	338	4450	852	5550	11190
BiLSTM, 2018 [72]	295	4130	317	5430	10172
DCNN, 2018 [73]	274	10400	284	12500	23458
Semi-supervised DL, 2019 [6]	231	3370	**251**	**2840**	6692
DAG network, 2019 [8]	229	2730	553	3370	6882
Multi-head CNN-LSTM, 2020 [7]	330	2880	401	6520	10131
AdaBN-DCNN, 2020 [5]	**220**	2250	260	3630	6360
The proposed method (A_4)	230	2556	325	3778	6889
The proposed method (A_{11})	247	**1743**	808	3051	**5851**

The first method (CNN) [3] is a conventional Feed-Forward NN with two convolutional layers. The following two rows prove that standard LSTM [4] and Bi-directional LSTM (BiLSTM) [72] outperform the CNN. The next method, DCNN [73], provide lower RMSE w.r.t. the methods using LSTM. The following four methods have been proposed more recently. In [6], a RNN-based deep architecture is used for RUL prediction under a semi-supervised setup, and a GA approach is used to tune its training hyper-parameters, rather than its architecture. The directed acyclic graph (DAG) network [8] is a variant of the CNN-LSTM architecture which employs a parallel path of CNN and LSTM to extract features. The following method is our previous work [7], which uses a handcrafted multi-head CNN-LSTM. Then, the DCNN with adaptive batch normalization (AdaBN) [5] achieves the lowest values in both metrics compared to the previous methods.

The last two rows of the tables report the experimental results obtained with our proposed method. Of note, by using ENAS-PdM the automatically discovered architectures give significantly better results w.r.t. those that we handcrafted in [7]. Most importantly, the proposed method outperforms any other DL-based method developed manually by human experts, in terms of aggregation of both metrics (with A_4 and A_{11} resulting, respectively, the best models w.r.t. sum of RMSE and sum of Score). Hence, our results considerably advance the state-of-the-art in RUL predictions.

To conclude our analysis, we provide some brief considerations about the training time of the best models found by our NAS approach. Although minimizing the training time was not one of the objectives of this work, we note that the best models discovered by ENAS-PdM are competitive, in terms of training time, with the state-of-the-art. In particular, we consider our best model (in terms of RMSE) found on FD001, namely A_4. To prove that this model can be trained with limited computation resources, we use Google Colab to measure its training time on FD001, and compare it with two of the best methods from the state-of-the-art, the DAG network [8] and AdaBN-DCNN [5]. The results of this comparison are shown in Table 16, where the training details and time of A_4 are compared with the details reported in the original papers proposing the DAG network and AdaBN-DCNN. Of note, the training time needed for the two compared methods is higher than 120 seconds, as opposed to approximately 80 seconds needed by our A_4 model. Notwithstanding this training time reduction, as we have seen our model can provide better RUL predictions.

Table 16. Training details and time of A_4 vs DAG network and AdaBN-DCNN on FD001.

Methods	Batch size	Epochs	Training time(s)
DAG network [8]	200	40	138.17
AdaBN-DCNN [5]	642	40	121.52
The proposed method (A_4)	400	20	**80.61**

6.1. Complexity

Finally, we discuss the time complexity of the proposed method. As seen, the multi-head CNN-LSTM is a DNN combining a multi-head CNN with an LSTM. Therefore, we first calculate the time complexity of the convolutional layers and that of the LSTM. After that, those two components are added up to analyze the time complexity of the CNN-LSTM architecture as a whole.

For h heads with C layers, the complexity of the convolutional layers can be defined as $O(C \cdot h \cdot m \cdot l_f \cdot o)$, where m denotes the number of convolutional filters, l_f indicates the length of the filter, and o indicates the size of the output feature map (using the same notation used in Section 4.1). Essentially, the complexity is determined by the convolution operation discussed in Section 3.1.

Considering the LSTM, its complexity is defined by $O(w)$, where w denotes the number of weights. The number of weights in an LSTM is defined as $4(U^2 + U + S \cdot U)$, where U denotes the number of units and S indicates the length of the input sequence to the LSTM. The DNN used in our study contains two LSTM layers with $U = 20 \cdot L_1$ and $U = 20 \cdot L_2$ units respectively. Moreover, S is the same as the number of segments k, defined as $\frac{l_s - l_w}{stride} + 1$. Based on the bounds of l_w, L_1 and L_2 shown in Table 1, the value of U, either $20 \cdot L_1$ or $20 \cdot L_2$, is then much larger than S. This indicates that the value of U^2 dominates all the other terms and determines the number of weights in the LSTM, i.e., $O(w) \approx O(4U^2)$. Hence, considering the fact that ${L_1}^2$ is much larger than ${L_2}^2$ for the worst case, the time complexity of the two stacked LSTM layers can be approximated as $O(c \cdot {L_1}^2)$, where the constant c is equal to $4 \cdot 20^2$.

Consequently, the complexity of the multi-head CNN-LSTM, per time step, is $O(C \cdot h \cdot m \cdot l_f \cdot o + c \cdot L_1^2)$. In the largest possible network, $c \cdot {L_1}^2 \gg C \cdot h \cdot m \cdot l_f \cdot o$, i.e., we can approximate the complexity per time step as $O(c \cdot L_1^2)$. Therefore, the complexity of the training process of the multi-head CNN-LSTM can be written as $O(c \cdot L_1^2 \cdot i \cdot e)$, where i and e denote the number of inputs and epochs respectively.

Finally, considering that the training is performed for each individual at each generation, the computational complexity of the whole NAS process is $O(c \cdot L_1^2 \cdot i \cdot e \cdot n_{pop} \cdot n_{gen})$, where n_{pop} is the size of the population and n_{gen} is the number of generations.

7. Conclusions

In this work, we presented a NAS approach that uses evolutionary search to tune the architecture parameters of a multi-head CNN-LSTM model specialized to make RUL predictions by processing multi-variate time series for PdM purposes. An important aspect of the proposed approach is its minimal computational cost. In many industrial contexts, there is usually no access to expensive computing infrastructures, but only limited computing resources are available. To make our approach suitable for such scenarios, we applied several mechanisms to shorten the NAS evaluation time and save computational cost. Firstly, we scheduled learning rate decay in such a way to shorten the evaluation time as well as to avoid overfitting. In addition, we used a history of the models evaluated during evolutionary process, to allow the algorithm to quickly converge to promising solutions without unnecessary evaluations.

By testing our approach on the C-MAPSS dataset we found that, in general, the best model discovered in one sub-dataset provides good results on other sub-datasets. When we compared the results of our proposed method with the methods from the state-of-the-art, we obtained relevant improvements in terms of aggregated RMSE and Score across the four sub-datasets. Without aggregation, we found that our approach is able to outperform the state-of-the-art in 3 out of 4 sub-datasets when using the RMSE. It is worth to note that by using evolutionary NAS we were able to find satisfactory CNN-LSTM architectures while exploring less than 4.3% of the total combinatorial search space. Overall, with our method the computation of one GPU for 8 to 18 hours is enough to obtain state-of-the-art RUL predictions. In fact, the resources available on free online services, such as Google Colab, or on cheap AWS cloud instances, also can afford to execute our algorithm.

Our work opens up several interesting research opportunities as to what concerns e.g. the use of network performance predictors, or the knowledge transfer across different datasets. In addition, a multi-objective optimization of both RMSE and Score can be an interesting direction to explore in the future.

Acknowledgements

This work was partially supported by Trentino Sviluppo. We also acknowledge the support of NVIDIA Corporation with the donation of the TITAN Xp GPU used for this research.

References

[1] W. Zhang, D. Yang, H. Wang, Data-driven methods for predictive maintenance of industrial equipment: A survey, IEEE Systems Journal 13 (3) (2019) 2213–2227. doi:10.1109/JSYST.2019.2905565.

[2] K. Kaiser, N. Gebraeel, Predictive maintenance management using sensor-based degradation models, IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans 39 (2009) 840 – 849. doi:10.1109/TSMCA.2009.2016429.

[3] G. S. Babu, P. Zhao, X.-L. Li, Deep convolutional neural network based regression approach for estimation of remaining useful life, in: Proceedings of the International Conference on Database Systems for Advanced Applications, Springer, 2016, pp. 214–228.

[4] S. Zheng, K. Ristovski, A. Farahat, C. Gupta, Long short-term memory network for remaining useful life estimation, in: Proceedings of the International Conference on Prognostics and Health Management, IEEE, 2017, pp. 88–95. doi:10.1109/ICPHM.2017.7998311.

[5] J. Li, D. He, A Bayesian optimization AdaBN-DCNN method with self-optimized structure and hyperparameters for domain adaptation remaining useful life prediction, IEEE Access 8 (2020) 41482–41501. doi:10.1109/ACCESS.2020.2976595.

[6] A. Listou Ellefsen, E. Bjørlykhaug, V. Æsøy, S. Ushakov, H. Zhang, Remaining useful life predictions for turbofan engine degradation using semi-supervised deep architecture, Reliability Engineering & System Safety 183 (2019) 240 – 251.

[7] H. Mo, F. Lucca, J. Malacarne, G. Iacca, Multi-Head CNN-LSTM with prediction error analysis for remaining useful life prediction, in: Proceedings of the Conference of Open Innovations Association, IEEE, 2020, pp. 164–171.

[8] J. Li, X. Li, D. He, A directed acyclic graph network combined with CNN and LSTM for remaining useful life prediction, IEEE Access 7 (2019) 75464–75475. `doi:10.1109/ACCESS.2019.2919566`.

[9] M. Canizo, I. Triguero, A. Conde, E. Onieva, Multi-head CNN–RNN for multi-time series anomaly detection: An industrial case study, Neurocomputing 363 (2019) 246 – 260. `doi:https://doi.org/10.1016/j.neucom.2019.07.034`.

[10] T. Elsken, J. H. Metzen, F. Hutter, Neural architecture search: A survey, Journal of Machine Learning Research 20 (2019) 1–21.

[11] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, X. Chen, X. Wang, A comprehensive survey of neural architecture search: Challenges and solutions, arXiv preprint arXiv:2006.02903.

[12] Y. Liu, Y. Sun, B. Xue, M. Zhang, G. G. Yen, A survey on evolutionary neural architecture search, arXiv preprint arXiv:2009.10937.

[13] K. Swersky, D. Duvenaud, J. Snoek, F. Hutter, M. A. Osborne, Raiders of the lost architecture: Kernels for Bayesian optimization in conditional parameter spaces, arXiv preprint arXiv:1409.4011.

[14] K. Kandasamy, W. Neiswanger, J. Schneider, B. Póczos, E. P. Xing, Neural architecture search with Bayesian optimisation and optimal transport, in: Proceedings of the International Conference on Neural Information Processing Systems, Curran Associates Inc., 2018, p. 2020–2029.

[15] H. Jin, Q. Song, X. Hu, Auto-Keras: An efficient neural architecture search system, in: Proceedings of the International Conference on Knowledge Discovery & Data Mining, ACM SIGKDD, 2019, pp. 1946–1956.

[16] K. Swersky, J. Snoek, R. P. Adams, Freeze-Thaw Bayesian optimization, arXiv preprint arXiv:1406.3896.

[17] A. Klein, S. Falkner, J. T. Springenberg, F. Hutter, Learning curve prediction with Bayesian neural networks, preprint.

[18] A. Zela, A. Klein, S. Falkner, F. Hutter, Towards automated deep learning: Efficient joint neural architecture and hyperparameter search, arXiv preprint arXiv:1807.06906.

[19] B. Baker, O. Gupta, N. Naik, R. Raskar, Designing neural network architectures using reinforcement learning, arXiv preprint arXiv:1611.02167.

[20] Z. Zhong, J. Yan, W. Wu, J. Shao, C.-L. Liu, Practical Block-Wise Neural Network Architecture Generation, in: Proceedings of the Conference on Computer Vision and Pattern Recognition, IEEE/CVF, 2018, pp. 2423–2432. `doi:10.1109/CVPR.2018.00257`.

[21] B. Baker, O. Gupta, R. Raskar, N. Naik, Accelerating neural architecture search using performance prediction, arXiv preprint arXiv:1705.10823.

[22] B. Zoph, Q. V. Le, Neural architecture search with reinforcement learning, arXiv preprint arXiv:1611.01578.

[23] H. Cai, T. Chen, W. Zhang, Y. Yu, J. Wang, Efficient architecture search by network transformation, in: Proceedings of the Conference on Artificial Intelligence, AAAI, 2018, pp. 2787–2794.

[24] H. Cai, J. Yang, W. Zhang, S. Han, Y. Yu, Path-level network transformation for efficient architecture search, in: Proceedings of the International Conference on Machine Learning, PMLR, 2018, pp. 678–687.

[25] B. Zoph, V. Vasudevan, J. Shlens, Q. V. Le, Learning transferable architectures for scalable image recognition, in: Proceedings of the Conference on Computer Vision and Pattern Recognition, IEEE/CVF, 2018, pp. 8697–8710.

[26] H. Pham, M. Guan, B. Zoph, Q. Le, J. Dean, Efficient neural architecture search via parameters sharing, in: Proceedings of the International Conference on Machine Learning, PMLR, 2018, pp. 4095–4104.
URL `http://proceedings.mlr.press/v80/pham18a.html`

[27] A. Ashok, N. Rhinehart, F. Beainy, K. M. Kitani, N2N learning: Network to network compression via policy gradient reinforcement learning, in: Proceedings of the International Conference on Learning Representations, PMLR, 2018, pp. 1–20.

[28] A. Brock, T. Lim, J. M. Ritchie, N. J. Weston, SMASH: One-shot model architecture search through hypernetworks, in: Proceedings of the International Conference on Learning Representations, PMLR, 2018, pp. 1–21.

[29] G. Bender, P.-J. Kindermans, B. Zoph, V. Vasudevan, Q. Le, Understanding and simplifying one-shot architecture search, in: Proceedings of the International Conference on Machine Learning, PMLR, 2018, pp. 550–559.
URL `http://proceedings.mlr.press/v80/bender18a.html`

[30] C. Zhang, M. Ren, R. Urtasun, Graph hypernetworks for neural architecture search, in: Proceedings of the International Conference on Learning Representations, PMLR, 2018, pp. 1–17.

[31] H. Liu, K. Simonyan, Y. Yang, DARTS: Differentiable architecture search, in: Proceedings of the International Conference on Learning Representations, PMLR, 2018, pp. 1–13.

[32] Y. Xu, L. Xie, X. Zhang, X. Chen, G.-J. Qi, Q. Tian, H. Xiong, PC-DARTS: Partial channel connections for memory-efficient architecture search, in: Proceedings of the International Conference on Learning Representations, PMLR, 2019, pp. 1–13.

[33] S. Xie, H. Zheng, C. Liu, L. Lin, SNAS: stochastic neural architecture search, in: Proceedings of the International Conference on Learning Representations, PMLR, 2018, pp. 1–17.

[34] F. P. Casale, J. Gordon, N. Fusi, Probabilistic neural architecture search, arXiv preprint arXiv:1902.05116.

[35] H. Cai, L. Zhu, S. Han, ProxylessNAS: Direct neural architecture search on target task and hardware, in: Proceedings of the International Conference on Learning Representations, PMLR, 2018, pp. 1–13.

[36] X. Chen, L. Xie, J. Wu, Q. Tian, Progressive differentiable architecture search: Bridging the depth gap between search and evaluation, in: Proceedings of the International Conference on Computer Vision, IEEE/CVF, 2019, pp. 1294–1303.

[37] J. Chang, Y. Guo, G. Meng, S. Xiang, C. Pan, et al., DATA: Differentiable architecture approximation, in: Proceedings of the International Conference on Neural Information Processing Systems, Curran Associates Inc., 2019, pp. 876–886.

[38] G. Li, G. Qian, I. C. Delgadillo, M. Muller, A. Thabet, B. Ghanem, SGAS: Sequential greedy architecture search, in: Proceedings of the Conference on Computer Vision and Pattern Recognition, IEEE/CVF, 2020, pp. 1620–1630.

[39] M. Zhang, H. Li, S. Pan, X. Chang, S. Su, Overcoming multi-model forgetting in one-shot NAS with diversity maximization, in: Proceedings of the Conference on Computer Vision and Pattern Recognition, IEEE/CVF, 2020, pp. 7806–7815.

[40] T. Elsken, J.-H. Metzen, F. Hutter, Simple and efficient architecture search for convolutional neural networks, arXiv preprint arXiv:1711.04528.

[41] L. Li, A. Talwalkar, Random search and reproducibility for neural architecture search, in: Proceedings of the Uncertainty in Artificial Intelligence Conference, PMLR, 2020, pp. 367–377.

[42] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, A. Kurakin, Large-scale evolution of image classifiers, in: Proceedings of the International Conference on Machine Learning, PMLR, 2017, pp. 2902–2911.
URL `http://proceedings.mlr.press/v70/real17a.html`

[43] M. Wistuba, Deep learning architecture search by neuro-cell-based evolution with function-preserving mutations, in: Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Springer, 2018, pp. 243–258.

[44] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, K. Kavukcuoglu, Hierarchical representations for efficient architecture search, in: Proceedings of the International Conference on Learning Representations, PMLR, 2018, pp. 1–13.

[45] L. Xie, A. Yuille, Genetic CNN, in: Proceedings of the International Conference on Computer Vision, IEEE/CVF, 2017, pp. 1388–1397. `doi:10.1109/ICCV.2017.154`.

[46] E. Real, A. Aggarwal, Y. Huang, Q. V. Le, Regularized evolution for image classifier architecture search, in: Proceedings of the Conference on Artificial Intelligence, AAAI, 2019, pp. 4780–4789.

[47] C. Saltori, S. Roy, N. Sebe, G. Iacca, Regularized evolutionary algorithm for dynamic neural topology search, in: Proceedings of the International Conference on Image Analysis and Processing, Springer, 2019, pp. 219–230.

[48] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, B. Hodjat, Evolving Deep Neural Networks, in: Artificial Intelligence

in the Age of Neural Networks and Brain Computing, Academic Press, 2019, pp. 293–312. doi:10.1016/B978-0-12-815480-9.00015-3.

[49] Y. Sun, B. Xue, M. Zhang, G. G. Yen, Completely automated CNN architecture design based on blocks, IEEE Transactions on Neural Networks and Learning Systems 31 (4) (2020) 1242–1254.

[50] Y. Sun, B. Xue, M. Zhang, G. G. Yen, J. Lv, Automatically designing CNN architectures using the genetic algorithm for image classification, IEEE Transactions on Cybernetics 50 (9) (2020) 3840–3854.

[51] Y. Sun, B. Xue, M. Zhang, G. G. Yen, Evolving deep convolutional neural networks for image classification, IEEE Transactions on Evolutionary Computation 24 (2) (2019) 394–407.

[52] D. Połap, An adaptive genetic algorithm as a supporting mechanism for microscopy image analysis in a cascade of convolution neural networks, Applied Soft Computing 97 (2020) 106824.

[53] H. Nguyen, X.-N. Bui, Soft computing models for predicting blast-induced air over-pressure: A novel artificial intelligence approach, Applied Soft Computing 92 (2020) 106292.

[54] J. R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, 1992.

[55] Y. Bi, B. Xue, M. Zhang, An evolutionary deep learning approach using genetic programming with convolution operators for image classification, in: Proceedings of the Congress on Evolutionary Computation, IEEE, 2019, pp. 3197–3204.

[56] B. Evans, H. Al-Sahaf, B. Xue, M. Zhang, Evolutionary deep learning: A genetic programming approach to image classification, in: Proceedings of the Congress on Evolutionary Computation, IEEE, 2018, pp. 1–6.

[57] D. J. Montana, Strongly typed genetic programming, Evolutionary Computation 3 (2) (1995) 199–230. arXiv:https://doi.org/10.1162/evco.1995.3.2.199, doi:10.1162/evco.1995.3.2.199.
URL https://doi.org/10.1162/evco.1995.3.2.199

[58] M. Suganuma, S. Shirakawa, T. Nagao, A genetic programming approach to designing convolutional neural network architectures, in: Proceedings of the Genetic and Evolutionary Computation Conference, ACM, 2017, pp. 497–504. doi:10.1145/3071178.3071229.
URL https://doi.org/10.1145/3071178.3071229

[59] M. Loni, A. Majd, A. Loni, M. Daneshtalab, M. Sjödin, E. Troubitsyna, Designing compact convolutional neural network for embedded stereo vision systems, in: Proceedings of the International Symposium on Embedded Multicore/Many-core Systems-on-Chip, IEEE, 2018, pp. 244–251.

[60] M. Suganuma, M. Kobayashi, S. Shirakawa, T. Nagao, Evolution of deep convolutional neural networks using Cartesian genetic programming, Evolutionary Computation 28 (1) (2020) 141–163. doi:10.1162/evco_a_00253.

[61] J. Miller, A. Turner, Cartesian genetic programming, in: Proceedings of the Genetic and Evolutionary Computation Conference – Companion, ACM, 2015, p. 179–198. doi:10.1145/2739482.2756571.
URL https://doi.org/10.1145/2739482.2756571

[62] F. Assunçao, N. Lourenço, P. Machado, B. Ribeiro, DENSER: deep evolutionary network structured representation, Genetic Programming and Evolvable Machines 20 (1) (2018) 5–35. doi:10.1007/s10710-018-9339-y.
URL http://dx.doi.org/10.1007/s10710-018-9339-y

[63] M. O'Neill, C. Ryan, Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language, Kluwer Academic Publishers, 2003.

[64] S. Bianco, M. Buzzelli, G. Ciocca, R. Schettini, Neural architecture search for image saliency fusion, Information Fusion 57 (2020) 89 – 101. doi:https://doi.org/10.1016/j.inffus.2019.12.007.
URL http://www.sciencedirect.com/science/article/pii/S1566253519302374

[65] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, L. Jackel, Handwritten digit recognition with a back-propagation network, in: D. Touretzky (Ed.), Advances in Neural Information Processing Systems, Vol. 2, Morgan-Kaufmann, 1990, pp. 396–404.

[66] J. B. Yang, M. N. Nguyen, P. P. San, X. L. Li, S. Krishnaswamy, Deep convolutional neural networks on multichannel time series for human activity recognition, in: Proceedings of the Conference on Artificial Intelligence, AAAI, 2015, p. 3995–4001.

[67] Y. Qian, P. C. Woodland, Very deep convolutional neural networks for robust speech recognition, in: Proceedings of the Spoken Language Technology Workshop, IEEE, 2016, pp. 481–488.

[68] G. Iacca, R. Mallipeddi, E. Mininno, F. Neri, P. N. Suganthan, Super-fit and population size reduction in compact differential evolution, in: Proceedings of the Workshop on Memetic Computing, IEEE, 2011, pp. 1–8.

[69] A. Saxena, K. Goebel, D. Simon, N. Eklund, Damage propagation modeling for aircraft engine run-to-failure simulation, in: Proceedings of the International Conference on Prognostics and Health Management, IEEE, 2008, pp. 1–9. doi:10.1109/PHM.2008.4711414.

[70] F. O. Heimes, Recurrent neural networks for remaining useful life estimation, in: Proceedings of the International Conference on Prognostics and Health Management, IEEE, 2008, pp. 1–6. doi:10.1109/PHM.2008.4711422.

[71] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, C. Gagné, DEAP: Evolutionary algorithms made easy, Journal of Machine Learning Research 13 (2012) 2171–2175.

[72] J. Wang, G. Wen, S. Yang, Y. Liu, Remaining useful life estimation in prognostics using deep bidirectional LSTM neural network, in: Proceedings of the Prognostics and System Health Management Conference, IEEE, 2018, pp. 1037–1042. doi:10.1109/PHM-Chongqing.2018.00184.

[73] X. Li, Q. Ding, J.-Q. Sun, Remaining useful life estimation in prognostics using deep convolution neural networks, Reliability Engineering & System Safety 172 (2018) 1 – 11.

Appendix A. Boxplots of the fitness distribution across generations for all the experiments

We report below the boxplots of the fitness distribution across generations for three trials of the proposed ENAS-PdM algorithm on each of the four C-MAPSS sub-datasets (FD001, FD002, FD003, and FD004) and both fitness functions (RMSE and Score). See the main text for details.

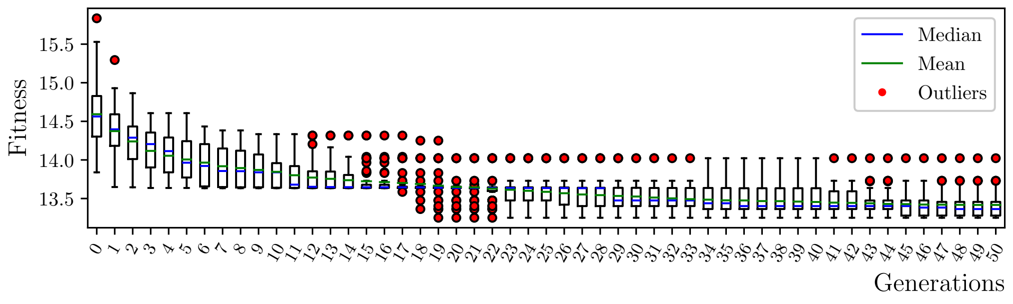


Figure A.7. Boxplots of the fitness distribution across generations: trial-1 on FD001, validation RMSE.

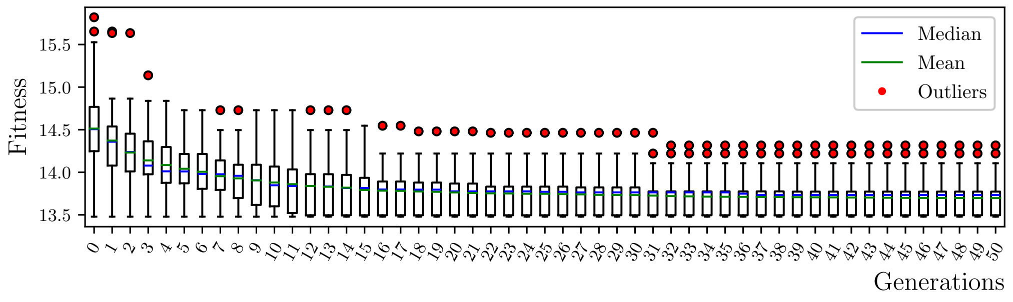


Figure A.8. Boxplots of the fitness distribution across generations: trial-2 on FD001, validation RMSE.

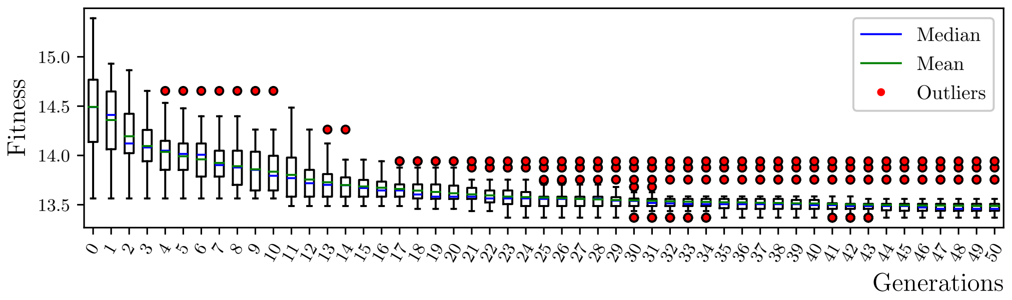


Figure A.9. Boxplots of the fitness distribution across generations: trial-3 on FD001, validation RMSE.

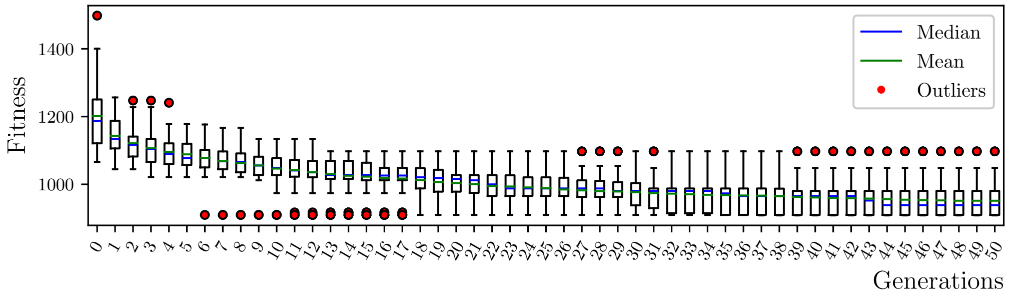


Figure A.10. Boxplots of the fitness distribution across generations: trial-1 on FD001, validation Score.

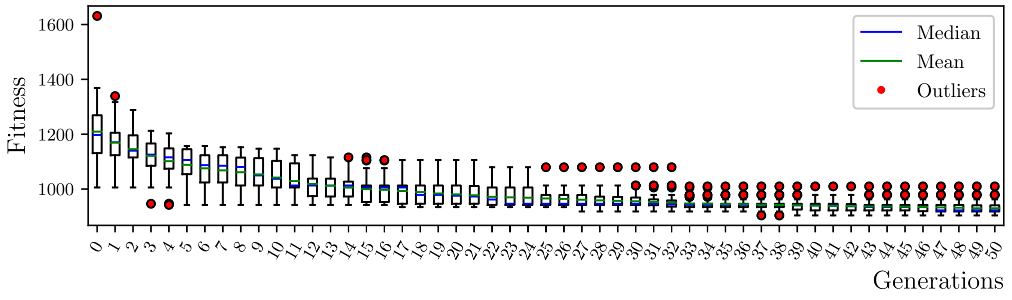


Figure A.11. Boxplots of the fitness distribution across generations: trial-2 on FD001, validation Score.

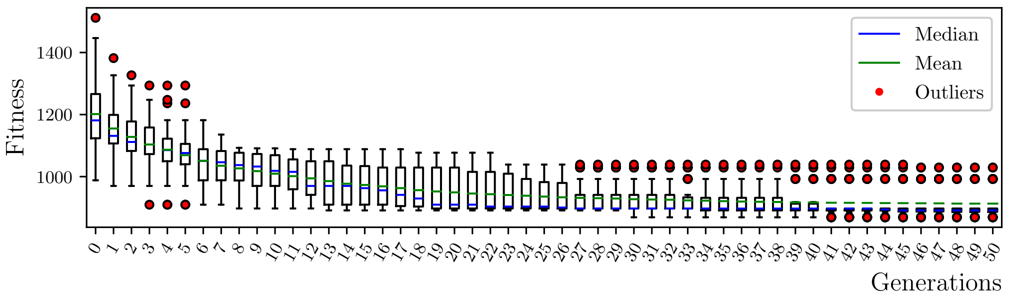


Figure A.12. Boxplots of the fitness distribution across generations: trial-3 on FD001, validation Score.

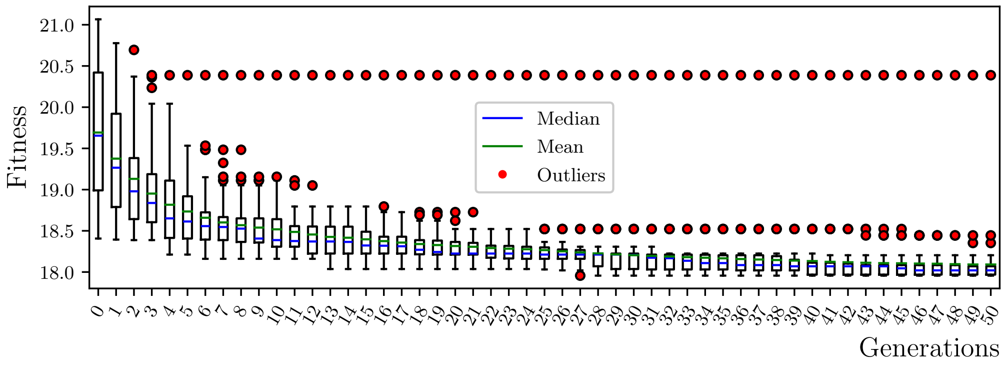


Figure A.13. Boxplots of the fitness distribution across generations: trial-1 on FD002, validation RMSE.

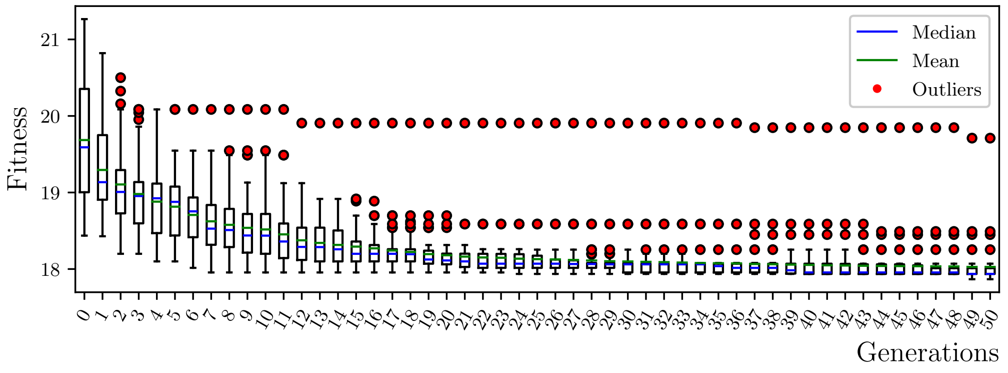


Figure A.14. Boxplots of the fitness distribution across generations: trial-2 on FD002, validation RMSE.

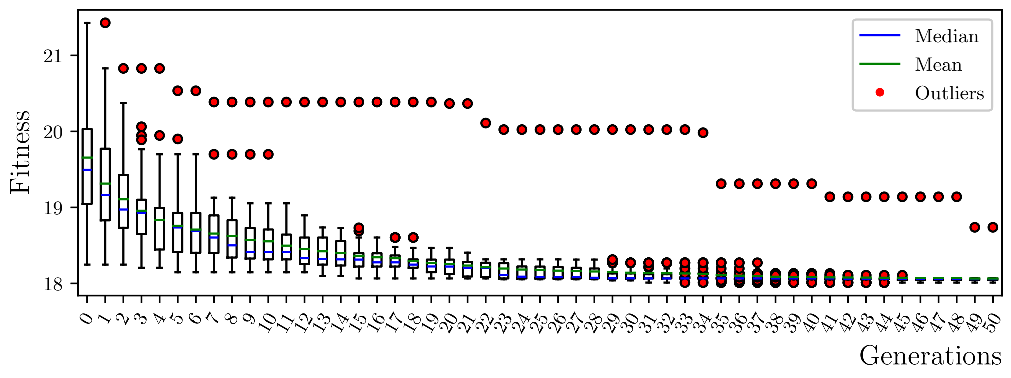


Figure A.15. Boxplots of the fitness distribution across generations: trial-3 on FD002, validation RMSE.

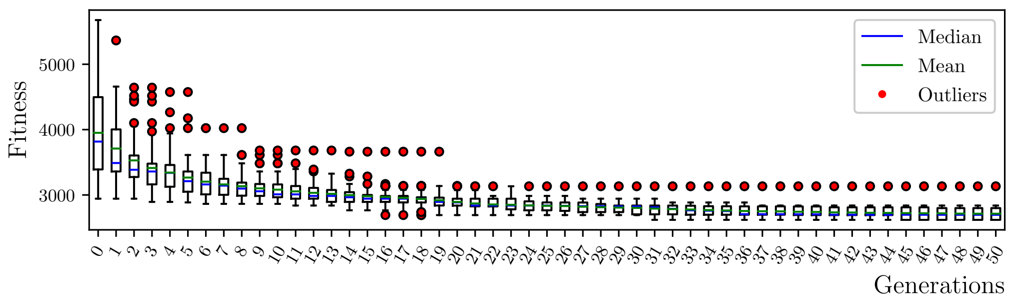


Figure A.16. Boxplots of the fitness distribution across generations: trial-1 on FD002, validation Score.

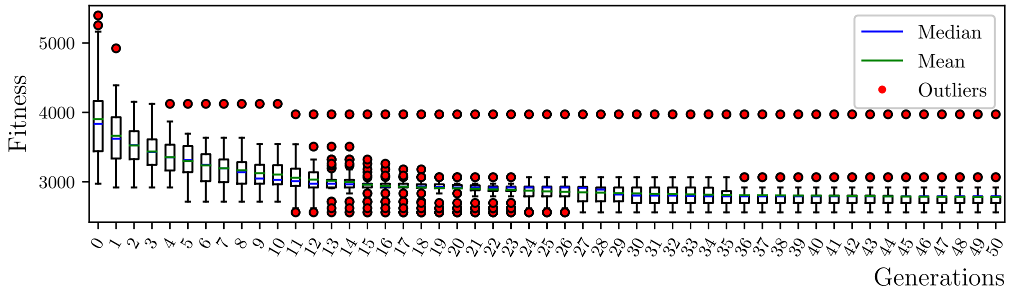


Figure A.17. Boxplots of the fitness distribution across generations: trial-2 on FD002, validation Score.

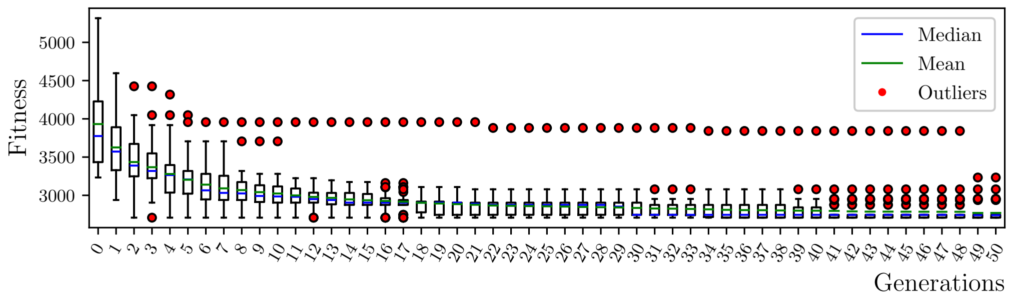


Figure A.18. Boxplots of the fitness distribution across generations: trial-3 on FD002, validation Score.

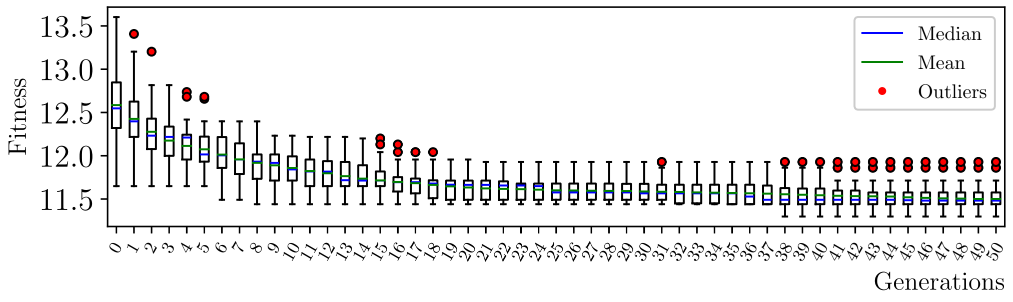


Figure A.19. Boxplots of the fitness distribution across generations: trial-1 on FD003, validation RMSE.

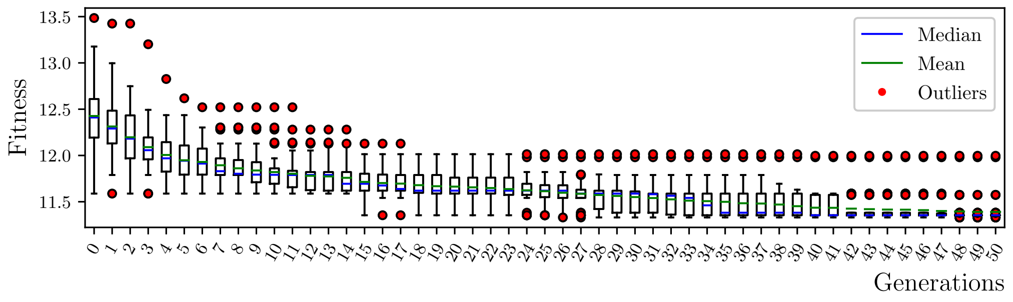


Figure A.20. Boxplots of the fitness distribution across generations: trial-2 on FD003, validation RMSE.

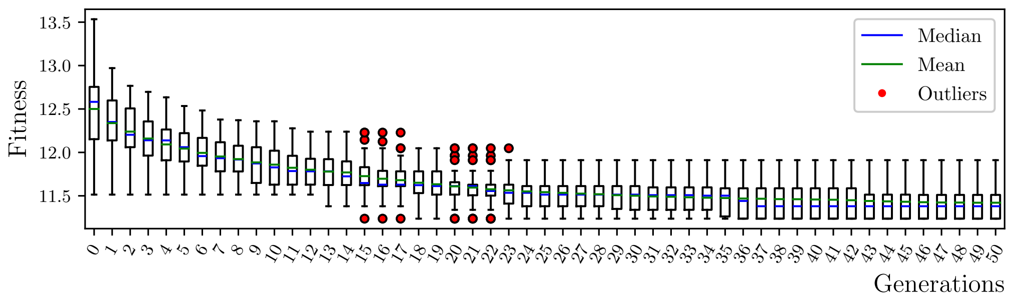


Figure A.21. Boxplots of the fitness distribution across generations: trial-3 on FD003, validation RMSE.

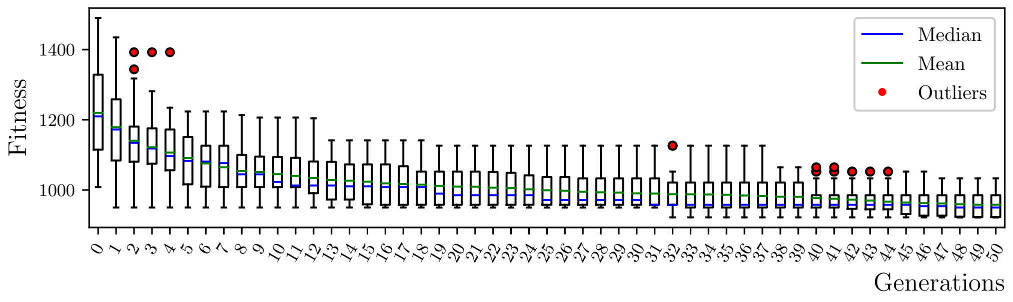


Figure A.22. Boxplots of the fitness distribution across generations: trial-1 on FD003, validation Score.

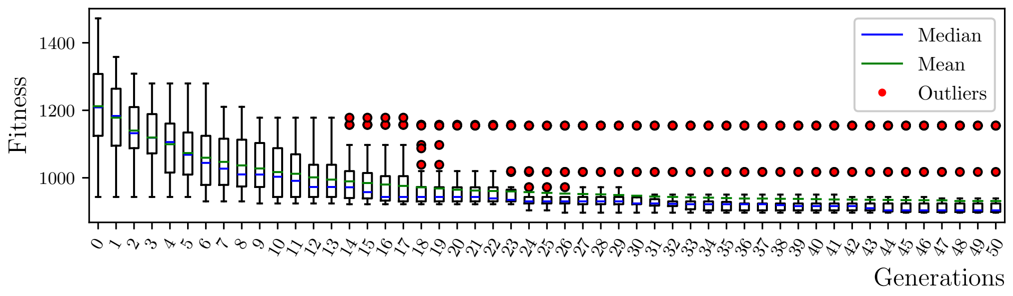


Figure A.23. Boxplots of the fitness distribution across generations: trial-2 on FD003, validation Score.

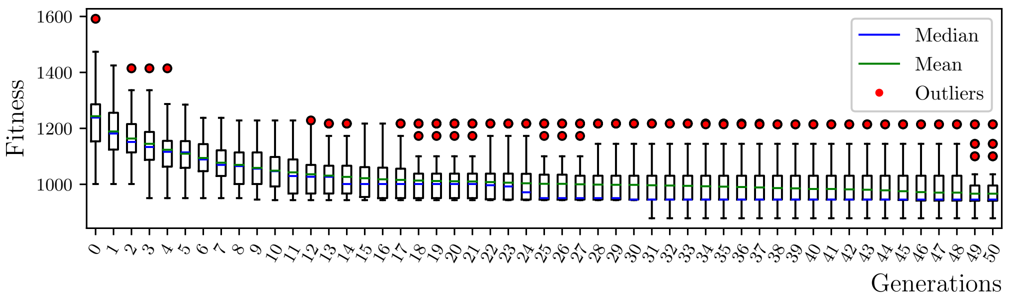


Figure A.24. Boxplots of the fitness distribution across generations: trial-3 on FD003, validation Score.

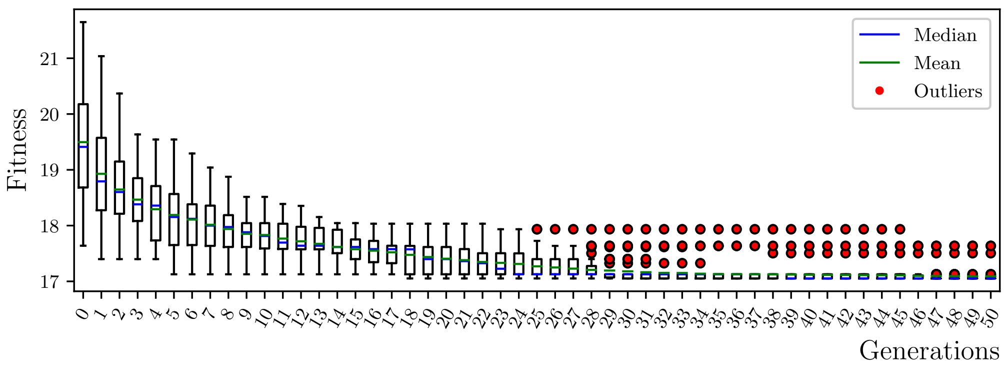


Figure A.25. Boxplots of the fitness distribution across generations: trial-1 on FD004, validation RMSE.

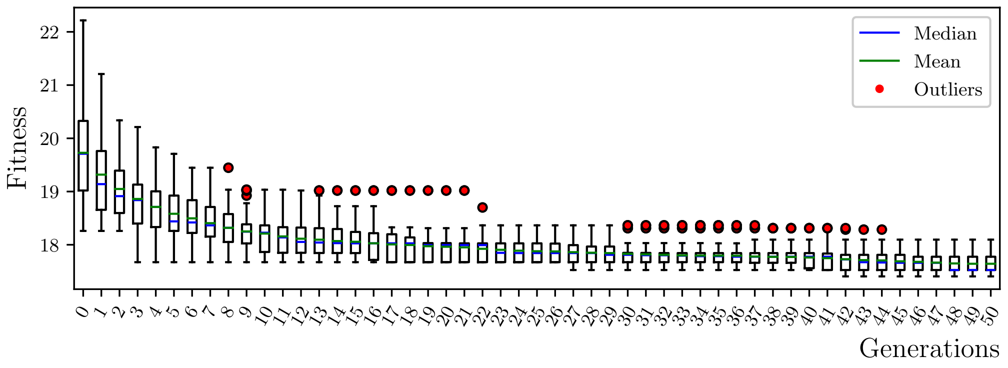


Figure A.26. Boxplots of the fitness distribution across generations: trial-2 on FD004, validation RMSE.

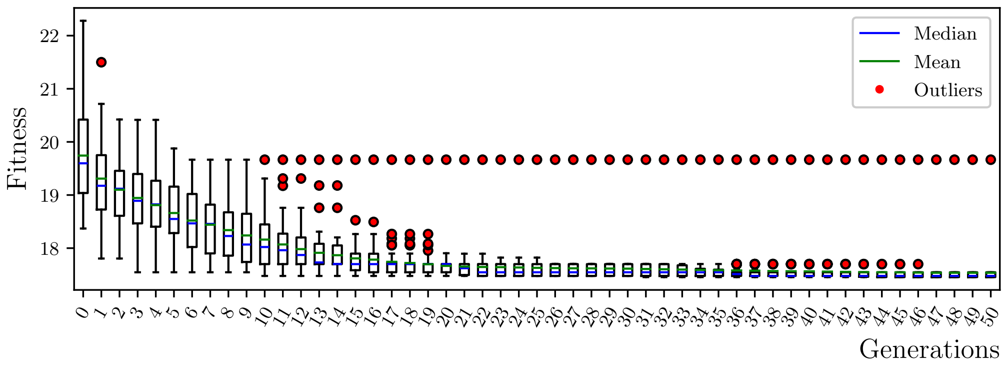


Figure A.27. Boxplots of the fitness distribution across generations: trial-3 on FD004, validation RMSE.

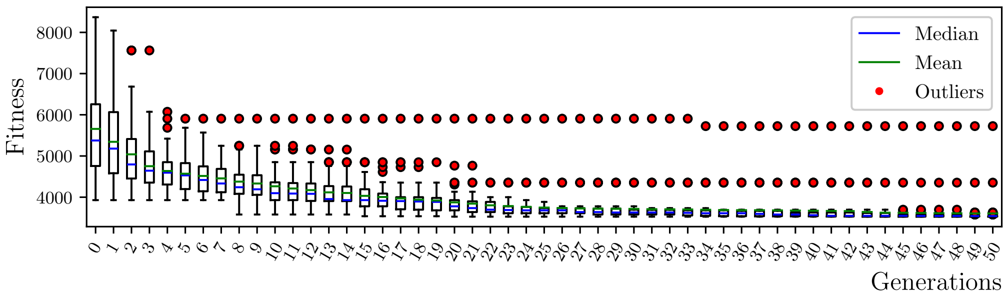


Figure A.28. Boxplots of the fitness distribution across generations: trial-1 on FD004, validation Score.

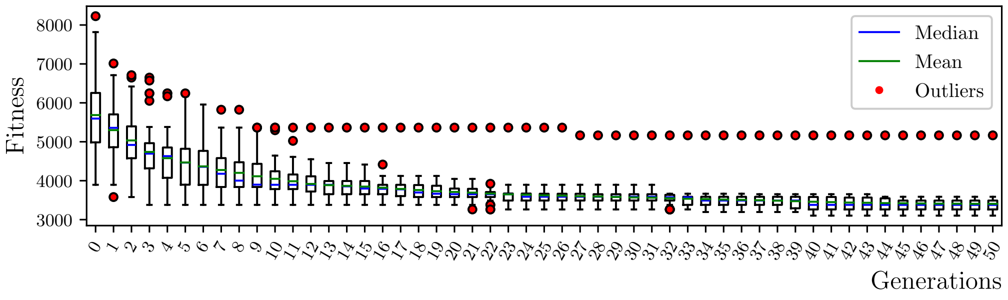


Figure A.29. Boxplots of the fitness distribution across generations: trial-2 on FD004, validation Score.

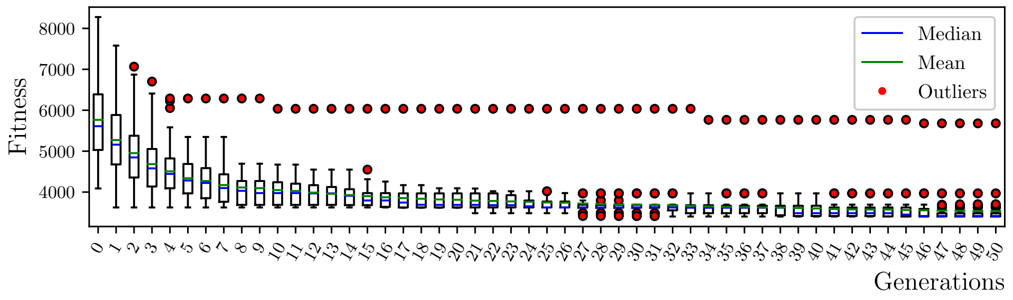


Figure A.30. Boxplots of the fitness distribution across generations: trial-3 on FD004, validation Score.