Accelerating Evolutionary Neural Architecture Search for Remaining Useful Life Prediction

Hyunho Mo[0000−0002−6497−2250] and Giovanni Iacca[0000−0001−9723−1830]

Department of Information Engineering and Computer Science,
University of Trento, Italy

Abstract. Deep neural networks (DNNs) obtained remarkable achievements in remaining useful life (RUL) prediction of industrial components. The architectures of these DNNs are usually determined empirically, usually with the goal of minimizing prediction error without considering the time needed for training. However, such a design process is time-consuming as it is essentially based on trial-and-error. Moreover, this process may be inappropriate in those industrial applications where the DNN model should take into account not only the prediction accuracy but also the training computational cost. To address this challenge, we present a neural architecture search (NAS) technique based on an evolutionary algorithm (EA) that explores the combinatorial parameter space of a one-dimensional convolutional neural network (1-D CNN) to search for the best architectures in terms of a trade-off between RUL prediction error and number of trainable parameters. In particular, a novel way to accelerate the NAS is introduced in this paper. We successfully shorten the lengthy training process by making use of two techniques, namely architecture score without training and extrapolation of learning curves. We test our method on a recent benchmark dataset, the N-CMAPSS, on which we search for trade-off solutions (in terms of prediction error vs. number of trainable parameters) using NAS. The results show that our method considerably reduces the training time (and, as a consequence, the total time of the evolutionary search), yet successfully discovers architectures compromising the two objectives.

Keywords: Evolutionary Algorithm · Multi-Objective Optimization · Convolutional Neural Network · Remaining Useful Life · N-CMAPSS.

1 Introduction

Predictive maintenance (PdM) is one of the key enabling technologies for Industry 4.0. It develops a maintenance policy using predictions of future failures of industrial components. Considering that this can be realized by estimating remaining useful life (RUL) of the target components, the RUL prediction has attracted considerable research interest, and much attention also from industry stakeholders.

Today, data-driven approaches using various deep learning (DL) models have gained increasing attention for developing RUL prediction tools. However, these

models are usually handcrafted and their performance depends on the network architecture, usually set empirically. Such a design process can be time-consuming and computationally expensive because of the needed trial-and-error. Neural architecture search (NAS), a technique that enables to design the architectures automatically, can be a reasonable solution for this problem. Particularly, the realization of NAS through an evolutionary algorithm (EA), the so called evolutionary NAS, has attracted considerable attention.

In the field of RUL prediction, a recent work [1] applied evolutionary NAS to design the architecture of a data-driven DL model automatically. The authors use a genetic algorithm (GA) to optimize the architecture of a complex DL architecture that was manually designed in their previous work [2], aimed at improving RUL prediction accuracy. Solving such an optimization problem for better prediction accuracy can be formalized as follows:

$$w^*(a) = \arg\min_{w} \mathcal{L}_{train}(w, a) \tag{1}$$

$$a^* = \arg\min_{a} \mathcal{L}_{val}(w^*(a), a) \tag{2}$$

where Eq. (1) describes an inner evaluation loop that aims to find the optimal weights w^* for a given architecture (described by its parameters a) w.r.t. the training loss $\mathcal{L}_{train}$, while the outer loop defined by Eq. (2) searches for the optimal architecture (i.e., the one described by the parameters a^*) w.r.t. the validation loss $\mathcal{L}_{val}$.

There are two problems in the above optimization task. As shown in Eq. (2), the algorithm searches for an optimal architecture w.r.t. the prediction accuracy, regardless of the size of the network. Although this aspect has not been thoroughly discussed so far in the existing literature, limiting the size of the network determined by the number of trainable parameters is an important objective in industrial contexts that normally seek to save cost by minimizing access to expensive computing infrastructures. To solve this problem, in this paper we propose to evolve a one-dimensional convolutional neural network (1-D CNN) simultaneously subject to the two objectives of reducing the RUL prediction error and minimizing the number of trainable parameters. For this multi-objective optimization (MOO) task, we use the well-known non-dominated sorting genetic algorithm II (NSGA-II) [3], which has already been applied successfully to NAS tasks [4, 5].

Another challenge of the aforementioned task is that it typically requires a lengthy and rather expensive training process. As shown in Eq. (1), evolutionary NAS is computationally expensive because each individual (i.e., candidate network architecture) should tune its parameters iteratively with gradient-based computations until convergence, before being evaluated on the validation data. To address this issue in our MOO approach, we propose a method for speeding up the training formulated in Eq. (1) by combining two techniques: architecture score without training [6] and extrapolation of learning curves.

The idea behind the architecture score is to predict the performance of a trained network from its initial state. This score measures the overlap of acti-

vations in an untrained network between different inputs from a mini-batch of data, so that a higher score at initialization implies better performance in terms of prediction error after training. Based on our preliminary experiments (not reported here for brevity), we found that this score is distinctive for networks with less than a certain number of trainable parameters. For those networks, we replace the expensive training step with the architecture score. For the networks with a larger number of trainable parameters, we instead apply extrapolation of learning curves. This technique prevents the need for full training (as done in the existing literature, where training is typically continued until a given maximum epoch, set large enough to allow convergence, before computing the validation loss), by training the network for a smaller number epochs, and observing the validation root mean square error (RMSE) after each training epoch. The observations are then used for estimating the validation RMSE at the maximum epoch. Specifically, we derive a learning curve based on the observations, and extrapolate it to take the predicted validation RMSE at the maximum epoch.

To test the proposed method, we have used the new commercial modular aero-propulsion system simulation (N-CMAPSS) dataset provided by NASA [7], which is a well-established benchmark in the area of RUL prediction. On this dataset, we search for optimal CNN architectures compromising the RUL prediction error and the number of trainable parameters. The experimental results verify that speeding up the evolutionary search causes the reduction of the hypervolume (HV) of just around 3% (compared to the NAS without acceleration), but the proposed method provides a considerable overall runtime reduction of approximately 75% in terms of GPU hours.

To summarize, the main contributions of this work can be identified in the following elements:

- The proposed method significantly shortens the evaluation time of the evolutionary NAS process.
- The networks discovered by the architecture search process represent successful trade-off between two conflicting objectives, namely the RUL prediction error and the number of trainable parameters.

The rest of the paper is organized as follows: in the next section Section 2, the general concepts on RUL prediction are introduced. The details of the proposed methods are presented in Section 3. Then, Section 4 describes the specifications of our experiments, while Section 5 presents the numerical results of the experiments. Finally, Section 6 discusses the conclusions of this work.

2 Background

Recently, various RUL prediction methods have been proposed, which can be mainly categorized into two approaches [8]: physics-based approaches and data-driven approaches. The former require extensive knowledge to analytically model the physical degradation process [9]. In practice, these implementations have been limited because the physics underlying degradation is well-understood only for relatively simple components, despite the huge amount of efforts [10]. On the

other hand, data-driven approaches do not suffer from the above problems as they assume that the information relevant to the health and lifetime of components can be learned from past monitoring data [11].

Due to their ability to learn degradation patterns directly from historical data without knowing the underlying physics, data-driven approaches have gained increasing attention. Especially, black-box models based on deep learning have been widely used for prediction [12]. Fig. 1 illustrates the flowchart of a data-driven RUL prediction task with a black-box model. The object of the RUL prediction is a target component. The sensors installed on the target collect the health monitoring data, usually recorded in the form of multivariate time series. The data are then fed into a black-box model that derives a RUL prediction as its output. This model is trained on historical data collected by run-to-failure operations. The training loss is then defined based on the difference between the predicted RUL and the actual RUL. After training, the model can directly provide the predicted RUL w.r.t. current sensor measurements. However, determining an appropriate black-box model is a key issue for developing successful data-driven RUL prediction tools.

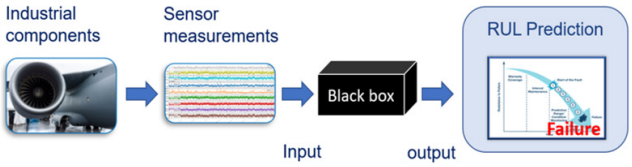

Fig. 1: Flow chart of a data-driven RUL prediction task.

Over the past decade, extensive research on data-driven approaches for RUL prediction using neural networks has been performed. One of the earliest works, introduced in [13], propose to use a multi-layer perceptron (MLP) for predicting the RUL of aircraft engines. The authors also propose to employ a convolutional neural network (CNN). Instead, the authors of [14] propose to a recurrent neural networks (RNN), in particular a long short term memory (LSTM), to recognize the temporal patterns in the time series. Considering the advantages of both CNNs and LSTMs, a combination of them has been used to predict RUL in [2]. As an alternative to use back propagation neural networks (BPNNs), Yang et al. [15] employ an extreme learning machine (ELM), a model originally introduced in [16], that achieves a much faster training compared to BPNNs. Recently, an autoencoder (AE) has been combined with RNNs [17] to obtain unsupervised learning. In [18], attention mechanism has been applied to a DL-based framework. Finally, deeper CNNs have been proposed in [10, 19], showing comparable performances to the aforementioned combined architectures.

3 Method

We present now the details of the proposed method: Section 3.1 describes the individual encoding and the optimization algorithm we used, while Section 3.2

explains how we defined and applied the two techniques for speeding up the evaluation.

3.1 Multi-objective optimization

Individual encoding Deep CNN architectures have provided outstanding performances on multivariate time series processing [20], also including RUL prediction [10, 19]. Therefore, we adopt a 1-D CNN as our backbone network, whose architecture should be optimized. This network consists of a set of 1-D convolution layers and one fully connected layer: the n_l stacked convolution layers aim to extract high-level feature representations, while the following fully connected layer uses all the extracted features for regression.

Fig. 2 visualizes a 1-D convolution layer in the network. Each of n_f filters of length l_f slides over its input features to apply convolution in the temporal direction. Each convolution layer is followed by an activation layer applying the rectified linear unit (ReLU) activation function. The feature map of the last convolution layer is flattened and fed into the fully connected layer comprising $n_{f.c.}$ neurons to predict the RUL.

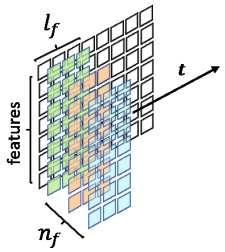

Fig. 2: Illustration of 1-D convolution layer with n_f filters of length l_f.

The number of convolution layers n_l and the two hyper-parameters regarding the convolution filter, n_f and l_f, contribute to the feature extraction. The number of neurons in the fully connected layer, $n_{f.c.}$, works on the regression task based on the extracted feature. All the four hyper-parameters largely affect the prediction error and determine the total number of trainable parameters in the network.

Based on the above description, we consider the optimization of the following architecture parameters:
- n_l, number of convolution layers;
- n_f, number of filters in each convolution layer;
- l_f, length of convolution filters;
- $n_{f.c.}$, number of neurons in the fully connected layer.

Considering that the architecture parameters are all integers, the encoding of solutions consists of four integers. The lower and upper bounds for each parameter considered in our evolutionary search are set as follows: $[3, 8]$ for n_l, $[5, 25]$

for both n_f and l_f, and $[5, 15]$ (multiplied by a fixed value of 10) for $n_{f.c.}$. These values have been chosen empirically. In particular, the smaller networks, which have too few trainable parameters, cannot decrease the training loss (i.e., they underfit), while the larger networks, containing too many parameters, may overfit the training data. Taking these two aspects into account, we set the bounds so to explore a parameter space of approximately 30,000 1-D CNN configurations. One additional note is that n_f is not valid for the last convolution layer. The number of filters in the layer is set to 5, to prevent the fully connected layer from receiving a too long flattened feature.

Optimization algorithm In order to optimize the architecture of the CNN described in Section 3.1, we use the well-known NSGA-II algorithm [3], to look explicitly for the best trade-off solutions in terms of RUL prediction error and number of trainable parameters. In the evaluation step of our evolutionary search, the fitness of each individual is calculated by generating a CNN (the phenotype) associated to the corresponding genotype, i.e., a vector containing the four parameters introduced in Section 3.1.

At the beginning of the evolutionary run, a population of n_{pop} individuals is initialized at random. In the main loop of the GA, an offspring population of the same size is generated by tournament selection, crossover and mutation. The new individuals are then put together with the parents. The combined population is then sorted according to non-domination. Finally, the best non-dominated sets are inserted into the new population until no more sets can be taken. For the next non-dominated set, which would make the size of the new population larger than the fixed population size n_{pop}, only the individuals that have the largest crowding distance values are inserted into the remaining slots in the new population. Subsequently, the next generation starts with the new population by creating its offspring population. We stop this loop after a fixed number of generations n_{gen}.

Regarding the genetic operators, we consider 1-point crossover with crossover probability p_{cx} set to 0.5 and uniform mutation with mutation probability p_{mut} set to 0.5. The probabilities have been chosen such that, in most cases, individuals are produced by either mutation or crossover (exclusively), so to avoid disruptive effects due to the combination of mutation and crossover that may lead to bad individuals. The expected number of mutations per individual is determined by the probability p_{gene}, set to 0.4. It indicates the probability of applying the mutation operator to a single gene. This means that we have, on average, 1.6 mutated genes out of 4, which allows us not only to have a relatively faster architecture search process, but also to avoid disruptive mutations.

Finally, we set n_{pop} and n_{gen} to 20 and 10 respectively. We have empirically found that these values allow enough evaluations to observe an improvement on the HV spanned by the discovered solutions. After 10 generations, the algorithm returns a Pareto front, that is defined as the set of trade-off solutions at the top dominance level.

3.2 Speeding up evaluation

Architecture score without training In almost all evolutionary NAS methods, the evaluation is the most time-consuming stage, because these methods typically evaluate a number of candidate networks on the validation data after the computationally expensive training [21]. In detail, a training set D_{train} is set to include all the available training data. This set is split into training purpose data, E_{train}, and validation purpose data, E_{val} (i.e., $D_{train} = E_{train} \cup E_{val}$). Then, E_{train} is used for the training process defined in Eq. (1) while the architecture search process, defined in Eq. (2), is based on E_{val}.

To reduce the time needed for the evolutionary search, we employ a speed-up technique called architecture score without training [6]. This method predicts the performance of a trained network based on its ability to discriminate between the different inputs of the network upon initialization, instead of training it.

Given that we use ReLU as the activation function in the networks, the output activation of each unit can indicate whether the unit is active or inactive; if the activation value is non-zero positive, the unit is active; otherwise, it is inactive. This is encoded as a binary bit, representing the former case as 1 and the latter as 0, i.e., we set the output activation of the non-zero positive case to 1. Given a data mini-batch $\mathrm{X} = \{x_i\}_{i=1}^{M}$, we feed an input sample x_i into a network containing N_{ReLU} activation units, and gather all the binary bits. Then, we obtain a binary code $\boldsymbol{c}_i \in \{0,1\}^{N_{ReLU}}$ for each sample x_i, thus in total we have M binary codes for the mini-batch.

The underlying intuition for the binary activation codes is that the similarity of two binary codes from two different inputs reveals how difficult it is to separate them for the network. For instance, if two different inputs have the same binary code, they lie within the same linear region defined by the activation function and therefore they are particularly difficult to distinguish [6].

The similarity between two different binary codes for x_i and x_j can be measured by the Hamming distance $d_H(\boldsymbol{c}_i, \boldsymbol{c}_j)$, and the correspondence between binary codes for X can be computed by the kernel matrix $\boldsymbol{K}_H$:

$$\boldsymbol{K}_H = \begin{bmatrix} N_{ReLU} - d_H(\boldsymbol{c}_1, \boldsymbol{c}_1) & \cdots & N_{ReLU} - d_H(\boldsymbol{c}_1, \boldsymbol{c}_M) \\ \vdots & \ddots & \vdots \\ N_{ReLU} - d_H(\boldsymbol{c}_M, \boldsymbol{c}_1) & \cdots & N_{ReLU} - d_H(\boldsymbol{c}_M, \boldsymbol{c}_M) \end{bmatrix}. \tag{3}$$

Based on $\boldsymbol{K}_H$, the architecture score is then defined as:

$$s = \frac{c}{\ln |\boldsymbol{K}_H|}. \tag{4}$$

Following Eq. (4), the determinant of the kernel matrix $|\boldsymbol{K}_H|$ is higher for the kernel closest to the diagonal, and large distances between two different codes mean that those can be well-separated by the neural network. Thus, a lower score for the same input batch at initialization implies a better prediction accuracy after training. We set the value of the constant c to 10^4, so that the score values in our work range approximately from 1 to 20.

Extrapolation of learning curves The evaluation step in evolutionary NAS typically requires training each network for a given number of epochs, with a relatively small learning rate. Following our previous work [1], we know that using a large learning rate enables to reduce the number of epochs, but it makes the validation curve fluctuate, thus providing unreliable evaluations caused by overfitting. Early stopping policy has been widely used for saving a few epochs, but its result is largely affected by how we define the performance improvement and the amount of patience. Moreover, the early stopping policy can lead to inaccurate performance estimations [21]. To mitigate this problem, we propose an extrapolation of learning curves that allows to save half of the training time w.r.t. a predetermined number of epochs. Note that here the learning curve is based on the validation RMSE across epochs.

Our basic approach is to derive the learning curve based on a set of functions $f(x)$, which are combined after fitting each of them to the observations. Specifically, we terminate the training at n_t epochs, that is half of the maximum epoch n_m planned for convergence, and collect all the validation RMSE for each of the n_t epochs. The observations are then used to fit each function defined in Table 1 by non-linear least squares minimization:

$$\text{minimize} \sum_{j=1}^{n_t} (y_j^o - f(x_j))^2$$

where y_j^o indicates the observed validation RMSE at x_j. The obtained function is denoted by f^*. The algorithm to solve the least squares problem is the Levenberg-Marquardt algorithm [22].

Table 1: Functions $f(x)$ used for extrapolation of learning curves. We chose a set of functions from the literature [23], whose shape coincides with our prior knowledge about the trend of the validation RMSE.

Name	Formula
MMF	$\alpha - \frac{\alpha-\beta}{1+\gamma x^\delta}$
Janoschek	$\alpha - (\alpha-\beta)e^{-\gamma x^\delta}$
Weibull	$\alpha - (\alpha-\beta)e^{-(\gamma x)^\delta}$
Gompertz	$\alpha + (\beta-\alpha)(1-e^{-e^{-\gamma(x-\delta)}})$
Hill custom	$\alpha + \frac{\beta-\alpha}{1+10^{(x-\gamma)^\delta}}$

As shown in Fig. 3, each curve drawn by f^* is close to the observation curve, but no single function can sufficiently describe the learning curve. Therefore, we combine all the obtained functions by solving a linear regression:

$$\text{minimize} \ \|\boldsymbol{F}^*\boldsymbol{a} - \boldsymbol{y}^o\|_2^2$$

where $\boldsymbol{F}^* \in \mathbb{R}^{n_t \times k}$ contains all the function values from k functions f^* (in our experiments, $k = 5$) for n_t epochs, and $\boldsymbol{y}^o \in \mathbb{R}^{n_t}$ is a vector of observations.

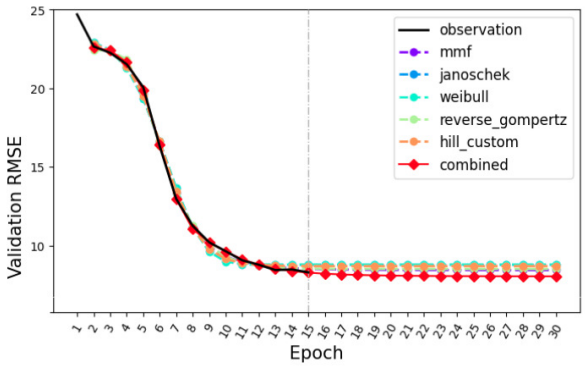

Fig. 3: An example of how the learning curve is derived from the $k = 5$ functions and the observations for 15 epochs. The red colored curve called "combined" represents the obtained learning curve. We take its value at 30 epochs and use it as the predicted validation RMSE.

The optimal $\boldsymbol{a} \in \mathbb{R}^k$, obtained by solving the linear problem problem, can be written as $\boldsymbol{a}^* = [a_1^*, \cdots, a_k^*]$. Our target value is the predicted validation RMSE at x_{n_m}. For that, first we take the function value at x_{n_m} for each function f^*, i.e., $\boldsymbol{f}^*(x_{n_m}) = [f_1^*(x_{n_m}), \cdots, f_k^*(x_{n_m})]$. The linear combination of these values with the weights $\boldsymbol{a}^*$ is then the target value $y_{n_m}^p$:

$$y_{n_m}^p = \boldsymbol{f}^*(x_{n_m}) \cdot \boldsymbol{a}^*. \tag{5}$$

If the validation RMSE has not converged yet, then our defined curve sufficiently decreases with x and the minimum observed value, $\min(\boldsymbol{y}^o)$, is greater than $y_{n_m}^p$. This decay can be defined as $d = \min(\boldsymbol{y}^o) - y_{n_m}^p$, where we take $y_{n_m}^p$ as the fitness value if the decay d is greater than 0.

During the evolutionary NAS process, many networks appear and each network converges at a different speed. Based on preliminary observations, we found that $y_{n_m}^p$ cannot be directly used as the fitness for some networks for which the learning curve decreases rapidly in the first few epochs and then reaches a plateau. In this case, our derived curve does not decrease with x and the decay d can be negative, while the actual validation RMSE may decrease even a little if we keep training the network. We compensate for this scenario by subtracting the absolute value of the decay to the minimum observed value. This way, we can assign a lower fitness value to the network that shows a validation RMSE trend that converges very quickly. Overall, the predicted fitness in terms of validation RMSE is then defined as:

$$fitness_{RMSE} = \begin{cases} y_{n_m}^p, & d > 0 \\ \min(\boldsymbol{y}^o) - |d|, & d \leq 0. \end{cases} \tag{6}$$

In our experiments, we set the maximum epoch n_m to 30, based on our previous works [1, 2, 5]. If we terminate the training too early (i.e., after less than half

n_m), then the predicted value may be too small because the learning curve shows no sign of convergence. On the other hand, using too many training epochs (close to n_m) would reduce any benefit of this speed-up technique. For these reasons, n_t is set to half n_m, i.e., 15.

4 Experimental setup

4.1 Computational setup and benchmark dataset

The 1-D CNNs are implemented using TensorFlow 2.4. All the experiments have been conducted on the same workstation with an NVIDIA TITAN Xp GPU, so that we can have a reliable comparison of the GA runtime in terms of GPU hours. To get reproducible results, we use the tensorflow-determinism library[1], which allows the DNNs implemented by TensorFlow to provide deterministic outputs when running on the GPU. The GA is implemented using the DEAP library[2]. Our code is available online[3].

To test the proposed method, we use the N-CMAPSS dataset [7] that consists of the run-to-failure degradation trajectories of nine turbofan engines with unknown and different initial conditions. The trajectories were generated with the CMAPSS dynamic model implemented in MATLAB, employing real flight conditions recorded on board of a commercial jet. Among the nine engines, we use 6 units (u_2, u_5, u_{10}, u_{16}, u_{18} and u_{20}) for the training set D_{train}, and the remaining 3 units (u_{11}, u_{14} and u_{15}) for the test set D_{test}. We select and use 20 condition monitoring signals following the setup in [10].

4.2 Data preparation and training details

As shown in Fig. 2, the DNNs used in our work require time-windowed data as an input to apply 1-D convolution in the temporal direction. To prepare the input samples for the networks, first each time series is normalized to $[-1, 1]$ by min-max normalization. Then, we apply a time window of length 50 and stride 50 so that the given multivariate time series consisting of the 20 signals is divided into input samples, with each sample of size 50×20. After slicing the time series into samples, we assign 80% randomly selected samples from D_{train} to E_{train}. The remaining samples in D_{train} are assigned to E_{val}, which is used for the fitness evaluation.

For training, we use stochastic gradient descent (SGD). In particular, AMSgrad [24] is used as optimizer after initializing weights with the $Xavier$ initializer. We set the initial learning rate to 10^{-4} and divide it by 10 after 20 epochs, following our previous observations on the effect of learning rate decay [1]. The size of the mini-batch for the SGD is set to 512. This size is also used for defining a mini-batch for the architecture score. We randomly choose 512 samples from

[1] `https://github.com/NVIDIA/framework-determinism`
[2] `https://github.com/DEAP/deap`
[3] `https://github.com/mohyunho/ACC_NAS`

E_{val}, and use it as the mini-batch X. On this regard, the ablation study in [6] verified that the choice of the mini-batch has little impact on the score trend over different network architectures.

5 Results

First, we generate 20 individuals (i.e., 1-D CNNs) randomly. Then, the multi-objective evolutionary process starts from the initial population. To perform a comparative analysis, we consider 5 different configurations w.r.t. the way of defining the fitness, denoted by $fitness_{RMSE}$: 1) using the architecture score, without training any networks; 2) using the validation RMSE, after training for 30 epochs; 3) using the validation RMSE, but training only for 15 epochs; 4) using the predicted validation RMSE at 30 epochs based on learning curve extrapolation, after training for 15 epochs; 5) using the architecture score if the network contains less than 5×10^4 trainable parameters, and the predicted validation RMSE with learning curve extrapolation otherwise.

The last configuration corresponds to our proposed method. We determine the decision threshold value to be 5×10^4 by analyzing the correlation between the number of trainable parameters and the architecture score. In Fig. 4, we can observe a negative correlation below the decision threshold. The difference in the architecture score for the range between 4 and 5 ($\times 10^4$) on the horizontal axis is trivial, but we take a large threshold value so that we can apply the architecture score based evaluation to as many networks as possible, because our major concern is to speed up the evolutionary search.

Here, we should note that the two different proposed surrogate mechanisms, i.e., the architecture score and the validation RMSE predicted by means of learning curve extrapolation, provide evaluation metrics that are obviously in different ranges. In order to use the score as fitness value (from the GA perspective), we

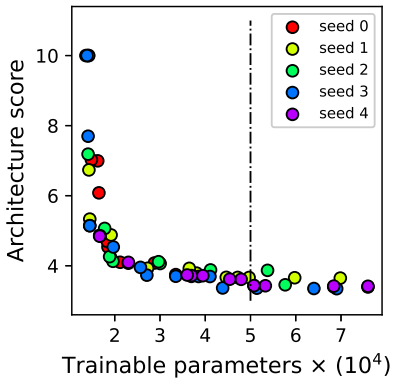

Fig. 4: Architecture score vs. number of trainable parameters on 100 randomly generated networks (20 for each seed). The dash-dotted line indicates the decision threshold.

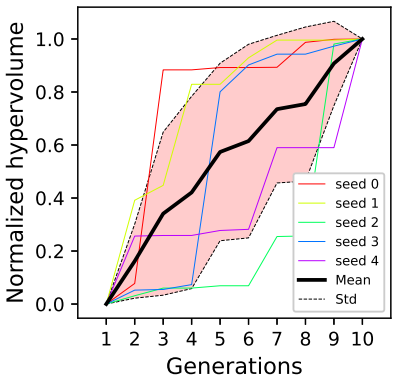

Fig. 5: Normalized validation HV across generations (mean ± standard deviation across 5 independent runs) for the proposed NAS approach.

Table 2: Summary of the comparative analysis for 5 different NAS configurations w.r.t. $fitness_{RMSE}$. The HV is an avg.±std. of the values in Fig. 6 that are based on the test RMSE and the number of trainable parameters. The boldface indicates the proposed method, which includes both architecture score and learning curve extrapolation. It gives the shortest GA runtime of all the methods that achieve better results than randomly generated solutions.

Methods (w.r.t. $fitness_{RMSE}$)	Test HV (avg.±std.)	GA runtime (GPU hours)
Initial population (without GA)	71.28 ± 0.95	-
Architecture score	70.26 ± 0.70	0.03 ± 0.01
Training 30 epochs	75.40 ± 0.55	4.96 ± 0.51
Training 15 epochs	72.94 ± 1.20	2.59 ± 0.30
Training 15 epochs + Extrapolation	73.81 ± 0.89	2.53 ± 0.15
Architecture score + Extrapolation	73.11 ± 0.58	1.23 ± 0.09

proceed as follows. For all the individuals in the initial population, we calculate both the architecture score and the actual validation RMSE value. Then, we fit a cubic function to these values, by means of least squares minimization, as explained in Section 3.2. This fitted curve is then used to convert, for any new network, the architecture score to the corresponding best fit validation RMSE value. This mechanism is meant to prevent any potential bias in the relative comparison of architectures evaluated by means of different metrics.

We execute 5 independent runs with different random seeds to improve the reliability of the results. While searching for the solutions, we consider the validation HV, which is calculated on the fitness space defined by the validation RMSE and the number of trainable parameters; we collect the validation HV across 10 generations, and normalize it to $[0, 1]$ by min-max normalization. The monotonic increase of the mean of the normalized validation HV in Fig. 5 indicates that the GA keeps finding new non-dominated solutions across the generations.

After finding the solutions, our result analysis is based on the test RMSE, which is evaluated a posteriori. Therefore, the HV in the rest of this paper is calculated on the space defined by the test RMSE and the number of trainable parameters. Fig. 6 shows the results of our experiments and Table 2 describes the summary of the comparative analysis. In the result analysis, we assess how the speed-up techniques affect the GA in terms of two metrics: 1) the quality of the solutions, represented by the HV, and 2) the GA runtime, in GPU hours.

It is obvious that the solutions based on the full training NAS are always the best in terms of HV, but it takes a rather long time (about 5 hours) to obtain them. When we merely use the architecture score without training, the NAS fails to find better solutions w.r.t. the initial population, because as said this approach alone cannot discriminate complex networks with a larger number of trainable parameters. If we terminate the training after 15 epochs, the obtained solutions are still better than the initial populations, but worse than the solutions obtained by training for 30 epochs. This implies that the learning curves of most of the networks appeared in our search converge later than 15 epochs. In this

case, our extrapolation technique helps find better solutions for the same 15 epochs training time, i.e., it improves the HV without significantly increasing the GA runtime. Finally, the proposed method, which combines the two techniques, further decreases the runtime while the HV slightly decreases. Its HV is not comparable to the HV obtained when training for 30 epochs, but this method allows to save a considerable amount of search time. Compared to the case of training for 15 epochs, the proposed method not only achieves better HV, but it saves more than 50% of GPU hours.

6 Conclusions

In this work, we presented a multi-objective evolutionary NAS approach that uses a custom GA to optimize the architecture parameters of a 1-D CNN specialized to make RUL predictions. The multi-objective optimization is based on NSGA-II and aims to achieve a trade-off between two competing objectives: the RUL prediction error and the number of trainable parameters. To improve the efficiency of evaluations in the NAS process, we introduced two acceleration methods for evaluating networks with either training for a reduced number of epochs or no training at all. The experimental results on the benchmark show that the speed-up techniques save about 75% of the GA runtime, while the solutions are slightly worse but still much better than randomly generated networks.

The most important limitation of this work is that the learning curve cannot fully simulate the actual learning trend for some networks. In future work, we can consider a variety functions (i.e., more than the 5 functions considered in this work), to enforce the learning curve decay for all the networks.

References

1. Mo, H., Custode, L., Iacca, G.: Evolutionary neural architecture search for remaining useful life prediction. Applied Soft Computing **108** (2021) 107474
2. Mo, H., Lucca, F., Malacarne, J., Iacca, G.: Multi-head CNN-LSTM with prediction error analysis for remaining useful life prediction. In: 2020 27th Conference of Open Innovations Association (FRUCT), IEEE (2020) 164–171
3. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation **6**(2) (2002) 182–197
4. Lu, Z., Whalen, I., Boddeti, V., Dhebar, Y., Deb, K., Goodman, E., Banzhaf, W.: NSGA-Net: neural architecture search using multi-objective genetic algorithm. In: Genetic and Evolutionary Computation Conference (GECCO). (2019) 419–427
5. Mo, H., Iacca, G.: Multi-objective optimization of extreme learning machine for remaining useful life prediction. In: International Conference on the Applications of Evolutionary Computation (Part of EvoStar), Springer (2022) 191–206
6. Mellor, J., Turner, J., Storkey, A., Crowley, E.J.: Neural architecture search without training. In: International Conference on Machine Learning, PMLR (2021) 7588–7598
7. Arias Chao, M., Kulkarni, C., Goebel, K., Fink, O.: Aircraft engine run-to-failure dataset under real flight conditions for prognostics and diagnostics. Data **6** (2021) 5

8. Atamuradov, V., Medjaher, K., Dersin, P., Lamoureux, B., Zerhouni, N.: Prognostics and health management for maintenance practitioners-review, implementation and tools evaluation. International Journal of Prognostics and Health Management **8**(3) (2017) 1–31
9. Bolander, N., Qiu, H., Eklund, N., Hindle, E., Rosenfeld, T.: Physics-based remaining useful life prediction for aircraft engine bearing prognosis. In: Annual Conference of the PHM Society. Volume 1. (2009)
10. Arias Chao, M., Kulkarni, C., Goebel, K., Fink, O.: Fusing physics-based and deep learning models for prognostics. Reliability Engineering & System Safety **217** (2022) 107961
11. Schwabacher, M., Goebel, K.: A survey of artificial intelligence for prognostics. In: AAAI fall symposium: artificial intelligence for prognostics, Arlington, VA (2007) 108–115
12. Khan, S., Yairi, T.: A review on the application of deep learning in system health management. Mechanical Systems and Signal Processing **107** (2018) 241–265
13. Babu, G.S., Zhao, P., Li, X.L.: Deep convolutional neural network based regression approach for estimation of remaining useful life. In: International Conference on Database Systems for Advanced Applications (DASFAA), Springer (2016) 214–228
14. Zheng, S., Ristovski, K., Farahat, A., Gupta, C.: Long short-term memory network for remaining useful life estimation. In: International Conference on Prognostics and Health Management (ICPHM), IEEE (2017) 88–95
15. Yang, Z., Baraldi, P., Zio, E.: A comparison between extreme learning machine and artificial neural network for remaining useful life prediction. In: Prognostics and System Health Management Conference (PHM). (2016) 1–7
16. Huang, G.B., Zhu, Q.Y., Siew, C.K.: Extreme learning machine: a new learning scheme of feedforward neural networks. In: International Joint Conference on Neural Networks (IJCNN). Volume 2., IEEE (2004) 985–990
17. Ye, Z., Yu, J.: Health condition monitoring of machines based on long short-term memory convolutional autoencoder. Applied Soft Computing **107** (2021) 107379
18. Chen, Z., Wu, M., Zhao, R., Guretno, F., Yan, R., Li, X.: Machine remaining useful life prediction via an attention-based deep learning approach. IEEE Transactions on Industrial Electronics **68**(3) (2021) 2521–2531
19. Li, X., Ding, Q., Sun, J.Q.: Remaining useful life estimation in prognostics using deep convolution neural networks. Reliability Engineering & System Safety **172** (2018) 1 – 11
20. Kiranyaz, S., Ince, T., Abdeljaber, O., Avci, O., Gabbouj, M.: 1-d convolutional neural networks for signal processing applications. In: ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE (2019) 8360–8364
21. Liu, Y., Sun, Y., Xue, B., Zhang, M., Yen, G.G., Tan, K.C.: A survey on evolutionary neural architecture search. IEEE transactions on neural networks and learning systems (2021)
22. Moré, J.J.: The Levenberg-Marquardt algorithm: implementation and theory. In: Numerical analysis. Springer (1978) 105–116
23. Domhan, T., Springenberg, J.T., Hutter, F.: Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In: Twenty-fourth international joint conference on artificial intelligence. (2015)
24. Kingma, D., Ba, J.: Adam: A method for stochastic optimization. In: International Conference on Learning Representations (ICLR). (2014)

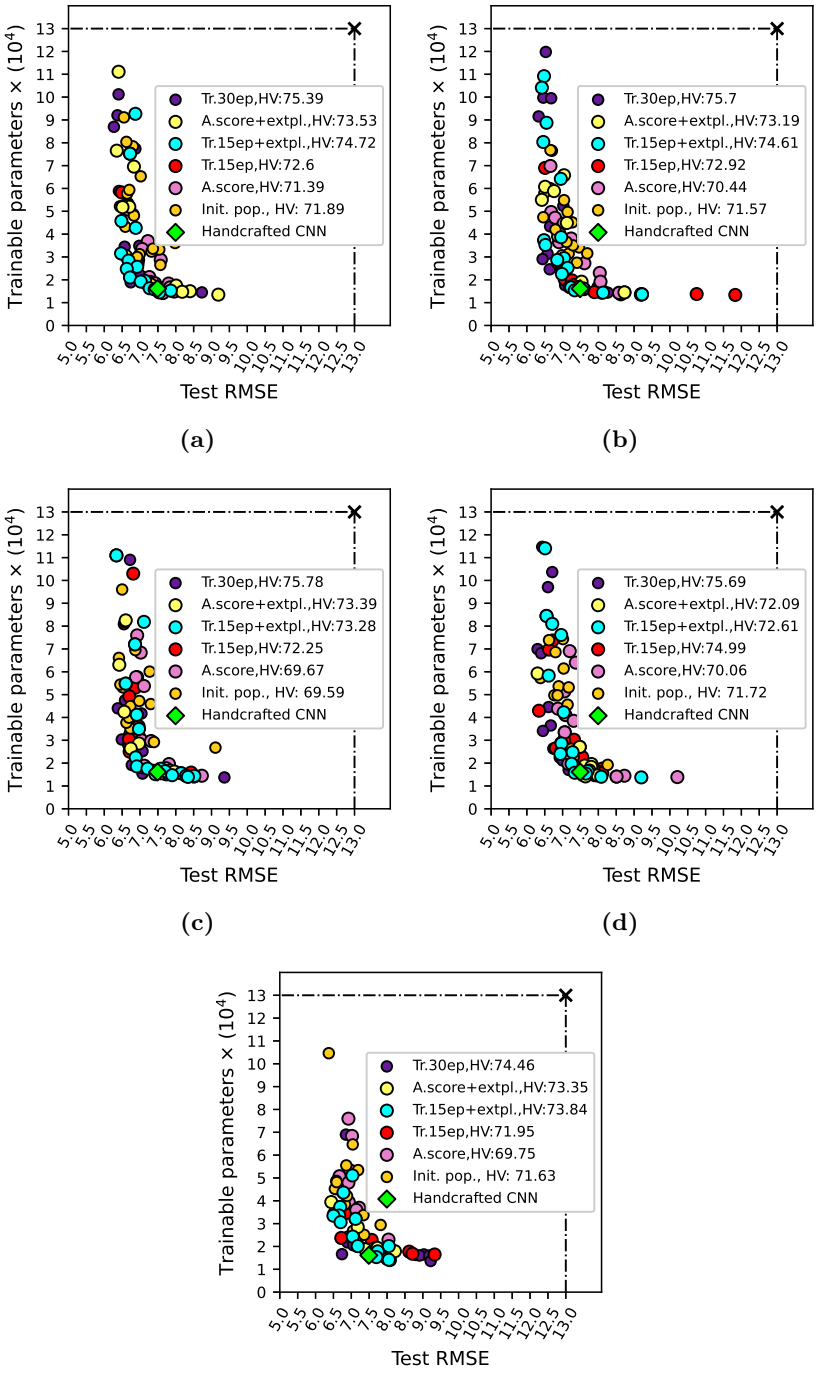

Fig. 6: Pareto front for 5 different NAS configurations w.r.t. $fitness_{RMSE}$: 30 training epochs ("Tr.30ep"); the combination of the architecture score and the learning curve extrapolation for ("A.score+extpl."); the learning curve extrapolation after 15 training epochs ("Tr.15ep+extpl."); 15 training epochs without extrapolation ("Tr.15ep"); merely using the architecture score ("A.score"). Each HV is calculated on the space shown in the figure which is defined by the test RMSE and the number of trainable parameters, and its value indicates the size of the space covered by the solutions of the corresponding configuration, with reference point (13, 13). Each figure shows the solutions found in 5 independent runs. The results of the handcrafted CNN, used as baseline, are taken from [10].