UNIVERSITÀ
DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE
DEPARTMENT OF INDUSTRIAL ENGINEERING
FONDAZIONE BRUNO KESSLER
Doctorate Program in Industrial Innovation

Evolutionary Optimization of Neural Architectures for Remaining Useful Life Prediction

Hyunho Mo

Advisor

Prof. Giovanni Iacca

Università degli Studi di Trento

Co-Advisor

Federico Lucca

BlueTensor S.r.l.

Abstract

Remaining useful life (RUL) predictions are a key enabler to achieving efficient maintenance in the context of Industry 4.0. Data-driven approaches, in particular employing deep neural networks (DNNs), have shown success in the RUL prediction task. However, although their architecture considerably affects performance, DNNs are usually handcrafted by human experts via a labor-intensive design process. To overcome this issue, we propose evolutionary neural architecture search (NAS) techniques that explore a search space using a genetic algorithm (GA). NAS automatically discovers the optimal architectures of neural networks for RUL predictions. Our GA allows an efficient search, finding high-quality solutions based on performance predictions which reduce the needed computational efforts for network training. In particular, first, we apply evolutionary computation to find the best architectures of deep and complex neural networks in terms of prediction accuracy. On the other side, we consider multi-objective optimization (MOO) of rather simple and fast neural networks to search for the best network architectures in terms of the trade-off between RUL prediction error and the number of trainable parameters, the latter being correlated with computational effort. In our experiments, we evaluate the performance of the found solutions on widely-used benchmark datasets, CMAPSS and N-CMAPSS. In comparison with the state-of-the-art, the obtained networks by our single objective NAS approach outperform other handcrafted recent DNNs in terms of prediction error, and the automatically designed networks by the multi-objective

NAS approach provide comparable results with manually designed traditional DNNs in terms of the test RMSE, but the number of trainable parameters is considerably smaller and the training time is significantly shorter. Our results demonstrate that the neural networks whose architecture is optimized by evolutionary NAS techniques can be a useful tool to solve the RUL prediction task.

Keywords

[Evolutionary algorithm, Neural architecture search, Deep learning, Predictive maintenance]

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Complex systems are at risk of eventual failure, and system failures can cause serious consequences in terms of safety issues as well as lead to high costs. It is crucial to prevent them through proper maintenance. Optimal maintenance enables the management of complex systems efficiently with low cost and effectively without failures.

The airline industry is a representative example considering that aircraft engines are very complex systems and their maintenance is closely related to not only cost, but also safety. To be specific, about 40% of the total maintenance cost is paid out as engine maintenance costs to comply with stringent safety requirements. This amount reaches roughly 20% of the total operating cost of flights[2]. As such, maintenance policies considerably affect cost and reliability.

1.1 Maintenance Policies

Maintenance policies can be categorized into three branches: reactive, aggressive, and proactive maintenance policies. The first branch is also called run-to-failure (R2F) maintenance and takes maintenance actions only after a failure has occurred. This primitive policy can only be applied to some systems in which failures never cause big consequences. On the contrary,

the second branch is needed for critical applications that do not allow any system failure. It is a very costly maintenance policy, but only safety issues are considered regardless of cost. The proactive maintenance policy is in the middle of the previous two, and it can be subdivided into scheduled and condition-based policies.

A scheduled maintenance policy refers to scheduling maintenance intervals in advance based on either operating time or usage so that the scheduled maintenance prevents failures. On the other hand, a condition-based maintenance policy indicates that we do not follow a prefixed schedule of maintenance but perform maintenance actions depending on the actual condition of the systems. It achieves just-in-time maintenance that takes actions not too early as well as not too late, while scheduled maintenance policy has the risk of failure occurring between maintenance intervals or the inefficiency of prematurely replacing parts that still have a large remaining useful life (RUL).

Condition-based maintenance policies are able to leverage either measured or calculated conditions. Measured conditions make it possible to track the measurable degradation of systems subject to maintenance. If it is not feasible to measure the conditions but these can be calculated, then condition-based maintenance policies require the development of physics-based models that represent physical failure mechanisms. The major disadvantages of utilizing them are that each model can always be representative only in a specific profile and the implementation of such a model takes large efforts of domain experts who have profound knowledge about the physics of failure. Physics-based approaches are still used especially when collecting monitoring or simulated data is difficult[3]. On the other hand, data-driven approaches are considered more useful when monitoring and/or simulated data are easier to acquire[4].

The availability of direct condition monitoring has increased today since it

becomes easier to collect condition-monitoring data via different sensors and store them. To utilize condition monitoring data and address the limitation of implementing physics-based models, data-driven approaches have received increased attention for developing optimal maintenance policies.

1.2 RUL predictions

Prognostics is an engineering subject that pays particular attention to the prediction of the time at which systems fall into a failure state. Prognostics and health management (PHM) is a computation-based paradigm that derives optimal maintenance policies by determining moments for maintenance based on the prediction of the RUL[5]. It aims to attain better planning of maintenance achieving low risk and minimal costs[6], and this goal can be realized by predicting the RUL accurately. In other words, solving RUL prediction tasks is a mainstream element of PHM. The prediction of RUL is usually referred to as prognostics. Prognostics is an engineering subject that pays particular attention to the prediction of the time at which systems fall into a failure state[7]. Namely, within the PHM framework, prognostics refers to predicting RUL of systems[5].

As discussed in Section 1.1, a variety of different sensors have been used to monitor the condition of complex systems these days, and each of them captures different physical properties representing different aspects of a system such as e.g., pressure, temperature, and vibration[6]. The collection of the sensor readings, big data for condition monitoring, can then be used to develop data-driven prognostic approaches.

Figure 1.1 outlines a data-driven RUL prediction[1] approach that employs a machine learning (ML) model, e.g. based on an artificial neural network

[1]In the literature, "RUL prediction" and "prognostics" are generically used to refer to the same discipline. Those are in general interchangeable, but we only use "RUL prediction" in the rest of this paper for consistency.

(ANN). The model is responsible for making the RUL prediction as its output from the condition measured by sensor readings. The training of the model exploits the historical data that comprise run-to-failure trajectories from different condition monitoring sensors. It aims to minimize the training loss where we derive the loss function from the difference between the predicted and the actual RUL (i.e., the ground truth that is assumed to be known for a set of training samples). During inference after training, the trained network outputs the predicted RUL for the given input. Eventually, precise RUL predictions given by the ML model can contribute to deriving optimal maintenance policies in the context of PHM. In fact, providing reliable RUL

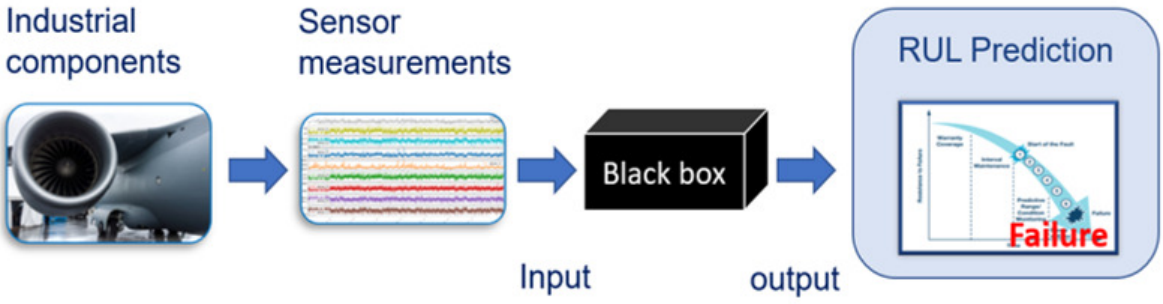


Figure 1.1: Flow of a data-driven RUL prediction task.

predictions allow users (e.g., plant owners or managers) to save expensive maintenance procedures, which often include costly inspection and imposed stops to the operation of the system. On the other hand, being able to anticipate, within a reasonable time window (i.e., neither too early nor too late) any potential system fault can not only enhance the quality of manufacturing processes and their output products but also cut losses caused by any unexpected downtime, even without regular and excessive maintenance to prevent failures. Providing accurate RUL predictions by means of the collected data (i.e., developing a data-driven RUL prediction model) is, however, a difficult task. In fact, data about faults are usually scarce (because these events typi-

cally occur less frequently than in normal operational conditions). Moreover, even when monitoring data about the system are available, these may not contain enough information to make the predictions (e.g., some states of the system that would be needed to predict faults may not be observable due to technical limitations or other kinds of practical constraints, such as lack of sensors).

1.3 Evolutionary NAS

Data-driven RUL prediction approaches can take advantage of supervised ML algorithms, which have grown rapidly in recent years. For instance, an end-to-end deep learning (DL) model can be used as a purely data-driven RUL prediction tool. This requires historical data on condition monitoring during run-to-failure operations. However, the data are very scarce because failures are usually prevented by the existing maintenance policies and we can merely get a limited number of cases where systems reach the end of life. Thus, from the given amount of run-to-failure data, finding an optimal ML model is a key challenge for developing a successful data-driven RUL prediction approach which is able to fully exploit the data.

In this context, ANNs, particularly deep neural networks (DNNs), have been widely used to recognize patterns in such limited data, and they have shown a large success in providing accurate predictions. Despite their promising performance in RUL predictions, a considerable amount of human effort is needed to develop them. The architecture of the networks can largely affect their performance, but it is hard to design even for experts who are specialized in both DL and RUL predictions. Such a design process corresponds to searching for suitable values of the parameters determining the architecture, and these values are often determined by empirical evidence or a trial-and-error process. This may not be an efficient way to build a RUL

prediction tool that fully exploits the capability of DL models.

To this end, different optimization techniques can be used to automate the design process. Instead of manually designing ANNs, we utilize optimization techniques to explore the search space and find optimal architectures. This is referred to as neural architecture search (NAS). Particularly, evolutionary computation has recently gained attention as a way to realize NAS, and it has been actually applied to develop different NAS techniques more recently[8]. It should be noted that we define the search space based on the pre-existing neural architecture types such as CNN-LSTM and Transformers. This is because DNNs having those certain types of neural architecture have already achieved success in various fields including the RUL prediction while variations of a specific type of architecture have been made by manually setting the value of design parameters. In other words, we take full advantage of the achievements made by the DL community w.r.t developing novel architecture types, while driving further advances in architecture search for the RUL prediction task. As such, the major motivation for our works is to present strategies to automatically determine architectures that provide promising performance compared to the architectures manually designed by human experts, when a particular type of network architecture is going to be used for the RUL prediction task.

Driven by the above motivation, we apply evolutionary computation to explore the combinatorial parameter space of different ANNs that are used as data-driven RUL prediction models. More specifically, we present genetic algorithm (GA) based neural architecture search (NAS) techniques that perform evolutionary optimization of neural architectures for RUL predictions and use them to find the optimal architectures of various data-driven network models such as a multi-head convolutional neural network with long short term memory (CNN-LSTM), the Transformer[9], an extreme learning machine (ELM)[10] network, a convolutional ELM (CELM)[11] and a one-

dimensional (1-D) CNN.

When we develop RUL prediction models based on NAS techniques, the following two different scenarios can be considered:

- finding the best architectures of deep and complex neural networks in terms of prediction accuracy;
- achieving a multi-objective optimization (MOO) of rather simple and fast neural networks where the two competing objectives are prediction accuracy and computational cost (which correlates to the number of trainable parameters).

Chapters 3 and 5 elaborate our contributions in the first scenario, and the descriptions in the chapter are based on the papers by Mo et al.[12,13]. We propose a custom evolutionary algorithm (EA) designed to find the best-performing deep network architectures in terms of prediction error. First, we use this evolutionary search to explore the combinatorial parameter space of a multi-head CNN-LSTM[12]. This algorithm is improved by incorporating a surrogate model into the evolutionary process, and it is used to optimize the architecture of Transformers used for RUL prediction[13].

Our research in the second scenario is detailed in Chapters 4 and 5 which explain our works presented in the papers by Mo et al.[14–16]. Since the above DNNs in the first scenario need an extensive computational effort to be trained, they may not fit industrial applications that allow using very limited computational resources. To provide proper solutions in this scenario, we consider a MOO technique that aims to find the best solutions achieving a trade-off between RUL prediction error and the number of trainable parameters. In detail, the well-known non-dominated sorting genetic algorithm II (NSGA-II)[17] is employed to discover the best solutions in the search space that comprises the hyper-parameters of an ELM network[14]. In addition to the vanilla ELMs, we also consider the architecture optimization of a CELM

network[15] which is the combination of a set of convolutional layers with random filters with a fully-connected layer trained by an ELM. Furthermore, we present the speed-up techniques that shorten the evaluation time of the evolutionary search process. It is then used for seeking non-dominated solutions from the combinatorial parameter space of a 1-D CNN[16].

When we focus only on minimizing prediction errors, we use the commercial modular aero-propulsion system simulation (CMAPSS)[18] dataset provided by NASA as a test case. The performance of the networks is evaluated via two different metrics: root-mean-square error (RMSE) and s-score. For the second scenario, we perform the experiments on the new CMAPSS (N-CMAPSS)[19] dataset which consists of a much larger amount of realistic data compared to the CMAPSS dataset (but one of the works[15] is evaluated on the CMAPSS). In the experiments, each run of our MOO algorithm gives a set of trade-off solutions (Pareto-front), and we evaluate the quality of the solutions by calculating either the performance for each of the two objectives or the hypervolume (HV).

1.4 Structure of the Thesis

The rest of the thesis is structured as follows: Chapter 2 introduces related works in data-driven RUL prediction. In Chapter 3, we present evolutionary computation-based NAS on DL models used for RUL predictions, so as to derive the best-performing DL-based RUL prediction method in terms of prediction error. The next chapter, Chapter 4, describes our work for the second scenario which requires optimizing ANNs w.r.t the two conflicting objectives of reducing the RUL prediction error while minimizing the number of trainable parameters. Chapter 5 details our experimentation of each method introduced in the previous two chapters and their results. Finally, Chapter 6 discusses the conclusions of our research.

Chapter 2

Related work

ANNs are computing systems composed of connected artificial neurons. They are inspired by the neurons in a biological brain. An artificial neuron transmits information by means of a signal which is a real number. For building an ANN, artificial neurons are aggregated into layers. The networks that have multiple layers are referred to as DNNs, and many ways of training them have been gradually developed aligning with the improvement of computing hardware performance[20]. DL indicates learning in DNNs, and it enables to extract from low to high-level features by leveraging its multi-layer architecture. It can be a subset of ML, and it has achieved remarkable success in various tasks such as object recognition[21].

Over the past decades, DL has developed more and more, and it has been used to address complex problems in various fields. In the field of computer vision (CV), deep CNNs have shown remarkable performances on various datasets[21,22]. Residual connections[23] involve skip connections on top of those CNNs, and they have contributed to advancing state-of-the-art performance. On the other side, a research question from another mainstream topic is how to improve computational efficiency without compromising performance. Variants of the traditional CNNs have been proposed as the solution, e.g., Inception architectured networks[24,25] which provide comparable

performance at a relatively low computational cost compared to the previous networks.

EA is a stochastic population-based metaheuristic. Based on the fact that biological systems result from an evolutionary process, EA attempts to copy the process of natural evolution, and it uses several bio-inspired mechanisms such as mutation and selection. One type of EA, GA, is used in optimization problems when any problem domain can have a suitable genetic representation and fitness function.

Developing the CNNs described above includes many parameters determining their architecture. Instead of relying on humans' choices regarding the layout design, evolutionary NAS techniques can be used to automate the design process. In particular, genetic algorithms (GAs)-based evolutionary search has been used to optimize the architecture of the CNNs and has shown promising results[26–29].

DL models are at the core of most state-of-the-art natural language processing (NLP) solutions for a wide variety of tasks. Different from traditional CV tasks such as image classification, NLP tasks commonly require dealing with long-term dependencies in sequential data. Long short-term memory (LSTM)[30] is able to handle them by keeping such dependencies between inputs in memory, and it is one of the most popular recurrent neural networks (RNNs) used for NLP tasks. A RNN model utilizing a gated recurrent unit (GRU)[31] instead of an LSTM unit has worked well and shown comparable results on sequence-based tasks. Particularly, encoder-decoder approaches have driven the recent trend in pure NNs-based machine translation[32]. More recently, a novel NNs handling long-term dependencies solely based on an attention mechanism, Transformers[9], have shown great success in various ML applications such as natural language understanding[33,34] and speech recognition[35,36].

Also for this research field employing DL models, significant effort is re-

quired to search for proper network architectures, and EAs have been used to automate the development process. In fact, EAs have been used to optimize neural networks for more than the past decades, by searching for both the optimal neural architectures and the weights of networks in the evolutionary process. Evolving NNs by means of EAs in this way is called neuroevolution[37], and there are various works that propose to use EAs to evolve DNNs, so-called deep neuroevolution[38,39]. The major difference between evolutionary NAS and neuroevolution is that EAs are used solely for finding optimal architectures in evolutionary NAS while they work for discovering both the neural architectures and the optimal weight values in neuroevolution. Both evolutionary NAS and neuroevolution have shown promising results in various tasks[8,40], but note that the strategies discussed in our works are based on evolutionary NAS.

For instance, evolutionary NAS techniques have been applied to intelligently choosing hyper-parameters of LSTM models[41,42] and a directed acyclic graph (DAG) network[43] for speech processing applications, aiming at improving their performance.

As discussed above, DL models hold state-of-the-art performance in CV and NLP. Those significant successes achieved so far in other fields have spurred various kinds of DL models to be actively used for data-driven RUL prediction tasks in PHM applications. One of the earliest works employs a 1-D CNN[44]. In particular, 1-D CNNs can benefit from recent progress in 2-D CNNs generally used for image data, since they can be developed by replacing 2-D kernels of 2-D CNNs with 1-D kernels. Thus, a number of papers have proposed to use 1-D CNNs as an RUL prediction tool[45–47].

RNNs can be applicable for processing sensor data which comprises multivariate time series. Therefore, existing works have introduced different RNNs used for RUL predictions[48–51]. Furthermore, they have been used in conjunction with CNNs; In combining of CNNs and RNNs, CNNs work for

extracting local features from sensor data, and RNNs contribute to recognizing temporal dependencies between the extracted features. This CNN-RNN structure has been widely adopted in PHM applications such as anomaly detection[52], fault diagnosis[53–55], and RUL prediction[56,57].

Very recently, autoencoder (AE) based models have been utilized to analyze temporal patterns on time series data[58,59], and encoder-decoder frameworks with attention mechanism have gained attention as a solution for RUL prediction tasks[60,61].

The RUL prediction problem can be addressed as a time series forecasting problem; each condition monitoring signal is first extended by using time series forecasting techniques[62], and the outcome can be used to estimate RUL prediction. DL has been used to address time series forecasting problems and various DL architectures[63] have been introduced; several works[64,65] employ neuroevolution to evolve NNs used for time series forecasting.

The DL models have indeed provided state-of-the-art results for various tasks given in PHM applications, while the DL moodels are human-designed networks that require a significant amount of human effort to choose the values of parameters determining their architecture design. Also, to our knowledge, evolutionary NAS techniques have been rarely used to develop DL-based RUL prediction models[8,66].

In this context, we hypothesize that evolutionary NAS can offer better-performing architectures of different DL models for the RUL prediction task, and the aim of our work is to validate the hypothesis based on evolutionary optimized neural architectures introduced in Chapters 3 and 4 and their experimental results described in Chapter 5.

Chapter 3

Evolutionary neural architecture search on deep learning models

In this chapter, we introduce two different works developing DL-based RUL prediction models based on evolutionary NAS. The first work applies a GA to optimizing the architecture of a multi-head CNN-LSTM. In the other work, we introduce a surrogate-assisted GA by incorporating a performance predictor in an evolutionary search process and apply it to finding optimal architectures of the Transformer[9]. The former was published in journal paper[12], and the latter has been published as journal paper[13].

3.1 Individual encoding

In this section, we describe the baseline structure of the multi-head CNN-LSTM and the Transformer, with details on the architecture parameters we optimize.

3.1.1 Multi-head CNN-LSTM

As discussed in Chapter 2, the CNN-RNN architecture can be a solution for various tasks in PHM applications[52–57]. Inspired by previous works, Canizo et al.[67] proposed to employ a multi-head CNN-RNN for anomaly detection

on the operating status of a service elevator made up of the following three parts: multiple parallel convolutional heads, a stack of recurrent layers, and a fully connected layer. While a typical 1-D CNN comprises a single path of stacked convolutional layers, a multi-head CNN consists of independent parallel paths in which each path comprises a CNN extracting local features from univariate time series.

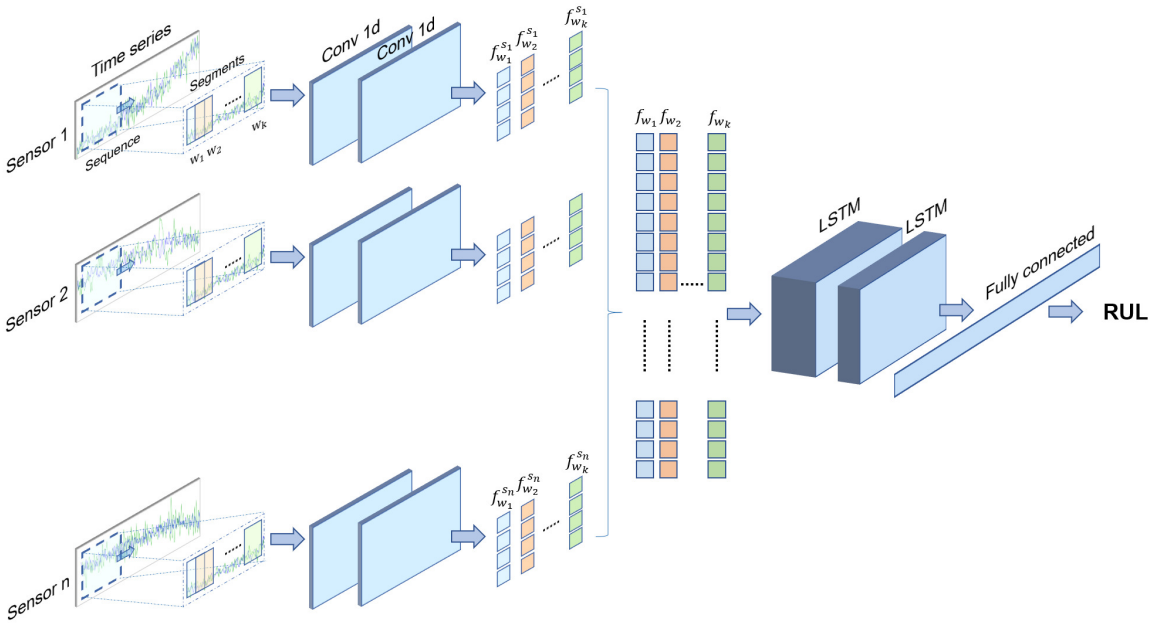


Figure 3.1: Multi-head CNN-LSTM architecture.

As described in Figure 3.1, the monitoring data of each sensor are segmented by a sliding window to generate fixed-size sequential data. Those are then respectively fed into each independent branch which is the so-called "head". The CNN in each head is responsible for extracting local features that appear in the univariate time series assigned to it. Different from a typical 1-D CNN, the convolutional layer in each head has its own kernel for individual convolution, without sharing parameters with other branches. At the end of the multi-head CNN, the extracted features from each head are concatenated. Thereafter, the following LSTM layers work for recognizing

the temporal dependencies from the concatenated features. Finally, the fully connected layer is fed by the output of the LSTM and produces a real value that corresponds to the predicted RUL.

The major advantage of such a parallel-branched architecture is that the learning of the convolutional filters in each head is specialized to handle a specific sensor. Considering this advantage, we employ a multi-head CNN-LSTM to predict RUL. The performance of the network largely relies on the architecture of the multi-head CNNs and LSTM. To fully exploit it, the CNNs in different heads may need to have different architectures so that each of them extracts proper features adapting to the particular sensor. Moreover, in a serial combination of multi-heads and LSTM, the architecture design of the independent CNNs and their extracted features largely affects the training of the following LSTM layers, which in turn has an impact on performance. To develop the sequential model properly, the design of the preceding and following parts cannot be independent, and it is obviously hard to analyze the relationship between them based on manual engineering such as trial-and-error approaches.

As shown in Figure 3.2, each head takes the input sequences extracted by applying a sliding window, and at the end of the CNN, it provides the extracted features. This convolutional feature extraction process from the sequential data includes many hyper-parameters. The length of a sequence is denoted by l_s. We slice each sequence into fixed-length *segments* and denote the length of a segment as l_w. The number of segments from a sequence is detnoed as k, and it is determined as follows: $k = \frac{l_s - l_w}{stride} + 1$.

The convolutional layer applies m filters of length l_f to each input, and it is followed by batch normalization and activation layers. The CNN in one head consists of C stacks of those layers, and it contains $k \cdot m \cdot C$ filters. Figure 3.2 shows that we can get k convolutional features as the output of the CNN for each head. The outputs of the n heads are then concatenated

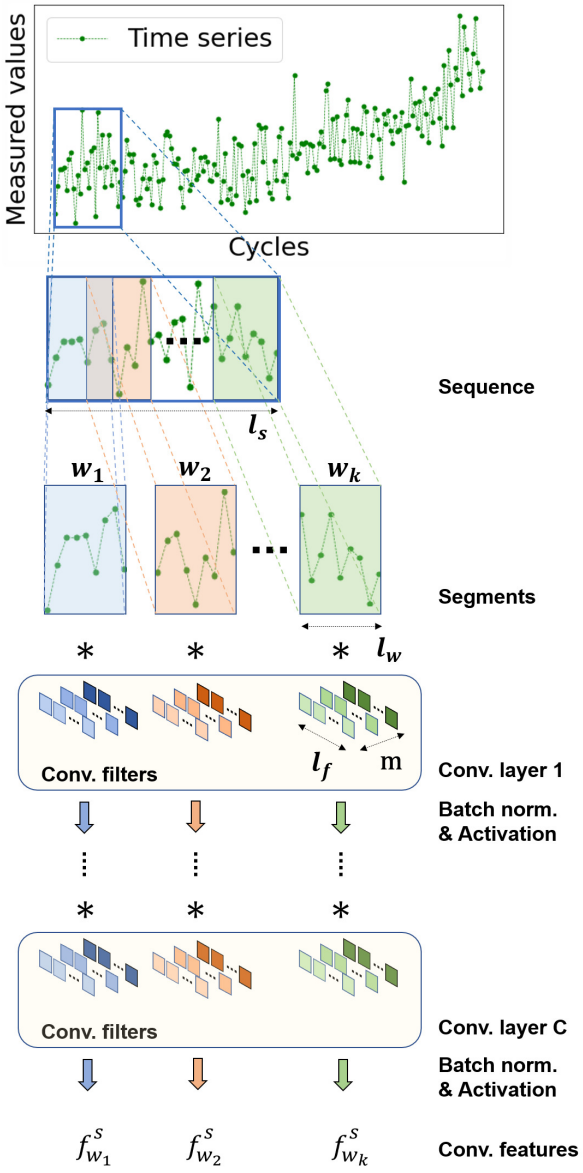


Figure 3.2: Visualization of how one head of the multi-head CNN extracts convolutional features from the given time series.

as shown in Figure 3.1, i.e., the length of each concatenated feature, l_c, is calculated as $l_c = l_w \cdot m \cdot n$.

Then, the concatenated features, $f_{w_1}, f_{w_2}, \ldots f_{w_k}$, are fed into the stacked LSTM layers, so that they generate an output based on the past information by considering long term dependencies between the features. The ability to

capture long-term dependencies of LSTM in general largely depends on the number of hidden units in it. Thus, we consider the number of hidden units in each of the two LSTM layers as hyper-parameters, $L_1 \cdot n_{lstm}$ and $L_2 \cdot n_{lstm}$ hidden units respectively, where n_{lstm} is a constant used as a multiplicand.

Under the parameterization introduced above, the parameters of the CNN and LSTM affect the performance of the network in a complex way. The length of segments l_w that we can choose controls the number of segments k. The span of the input data for the following LSTM is then determined by k. Furthermore, l_w can affect the decision on the other hyper-parameters in the CNN as well, since the required number of filters m and filter length l_f are different for proper feature extraction depending on the input size.

Table 3.1: Architecture parameters of the multi-head CNN-LSTM and their bounds.

Parameter	Description	Min	Max
l_w	length of segments	1	5
l_f	length of convolutional filters	1	5
m	number of convolutional filters	1	10
C	number of convolutional layers	1	2
L_1	multiplier of hidden units number (1st LSTM layer)	4	20
L_2	multiplier of hidden units number (2nd LSTM layer)	4	15

As such, the optimal value for one parameter depends on the values of the other parameters. This makes it difficult to choose a set of values based on empirical evidence; it is time-consuming and nearly unfeasible to manually examine the parameters effect on the network performance by a trial-and-error process. To this end, we use evolutionary computation to exploring such a combinatorial parameter space to find the optimal architecture of the multi-head CNN-LSTM network used for RUL predictions.

The search space is defined by the architecture parameters listed in Table 3.1. The upper bound of segment length l_w is set based on the sequence length and our previous knowledge[68]. It is the same as the upper bound of

l_f, because the filter size does not need to exceed the input size. Regarding L_1 and L_2, we estimate the proper range based on our previous work[68]; when considering less than 80 LSTM hidden units, the networks are prone to underfitting, while they can suffer from overfitting if the number of hidden units is larger than 400. Because this range is too large to be explored, we divide it by a constant value of 20 when we derive our genotype representation which is a list of integers within the bounds. Considering that the second LSTM layer is used to get high-level abstraction from the output of the preceding layer, we set the upper bound of L_2 to 15 which is smaller than that of L_1. The upper bound of the remaining parameters is determined by considering the memory size of the computational resource used in our experiments.

Based on the individual encoding described above, an individual of the EA is an integer vector containing the parameters described in Table 3.1, i.e., our genotype representation is $[l_w, l_f, m, C, L_1, L_2]$. When we introduce an example of an individual that comes out as $[2, 2, 5, 1, 10, 5]$, its phenotype is then a CNN-LSTM network that has the following architecture: segment length 2 and 1 convolutional layer comprising 5 convolutional filters of length 2 followed by stacked LSTM with 200 and 100 hidden units respectively. For all the individual encoding introduced in Chapters 3 and 4, the corresponding phenotype of an individual is a constructed network in this manner.

3.1.2 Transformers

The Transformer is a novel straightforward end-to-end model that can handle long-term dependencies in sequential data solely relying on an attention mechanism. Even dismissing either RNNs or CNNs, it has shown large success in various ML applications, such as CV[69,70], speech recognition[35,36], and NLP[33,34].

Considering that the Transformer has originally been established as a sequence transduction model, it has also been used for several tasks dealing

with time series data: anomaly detection[71], time series forecasting[72], time series regression and classification[73]. Another advantage of the Transformer is that it does not involve RNN modules which have been widely used to capture temporal patterns present in sequences[48,68], but several limitations; RNNs from gradient vanishing problems, and even networks with these problems solved such as LSTM or gated recurrent unit (GRU)[31] are computationally expensive to be trained.

In spite of many successful applications based on the advantages, the Transformer has not been fully exploited since it is usually handcrafted for each application; its architecture is determined by means of experience or manual observation regarding the relationship between architecture variation and change in performance. To this end, we propose to automatically design the Transformer using GA to have better architectures for a given task. Specifically, we consider the architecture of the Transformer comprising a parallel encoder structure[74] as a backbone to be optimized by our GA.

In the following, we formulate the RUL prediction task and describe the background concepts of the Transformer network.

Our base model described above involves many parameters that can vary the architecture of the Transformer network. Variations on the Transformer architecture can cause changes in performance[9], and their effect is typically investigated empirically, by changing one parameter at a time[9,75]. However, this is a labor-intensive design process, and the empirical evidence obtained in this way does not reflect the complex dependencies between the architecture parameters.

Different from the naïve approach, here we consider the Transformer architecture parameter choice as an optimization problem, aiming to obtain the maximum possible performance by systematically diversifying all the architecture design parameters. To this end, we use evolutionary optimization.

In this case, each "individual" in the population handled by the evolutionary algorithm (EA) encodes a candidate solution for the Transformer network design problem, i.e., a vector representation of all the parameters that can vary the Transformer architecture.

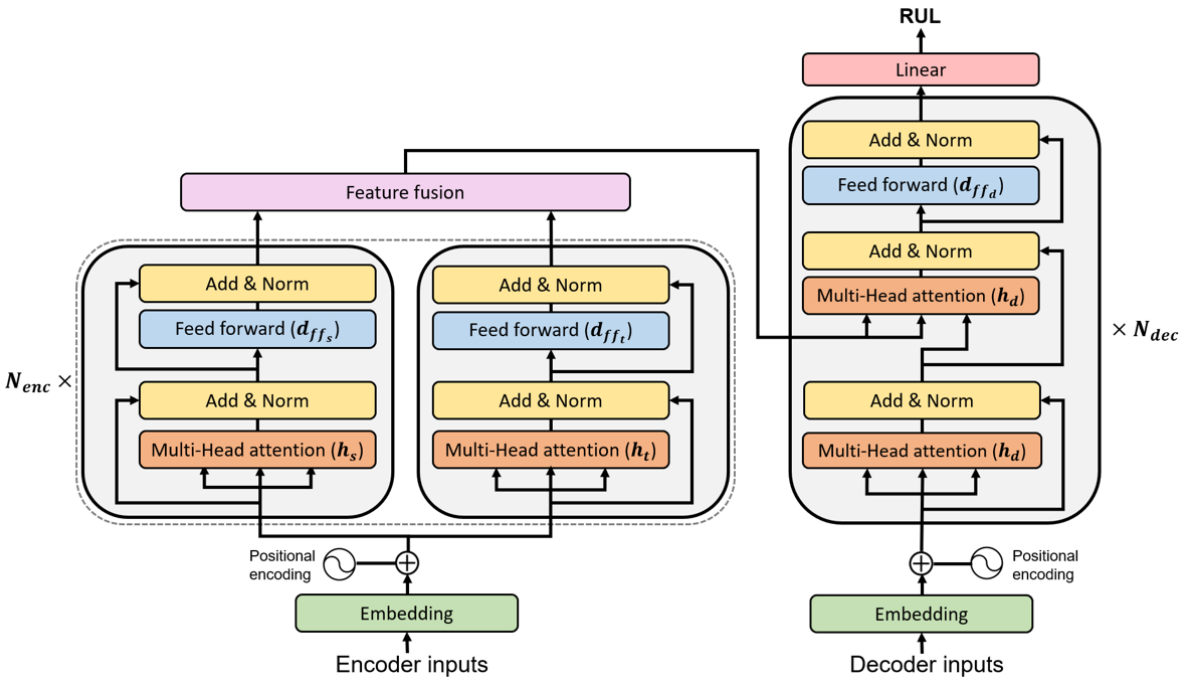


Figure 3.3: Model architecture of the Transformer.

The Transformer follows the encoder-decoder architecture drawn in Figure 3.3. Both the encoder and the decoder are made up of a multi-head attention layer followed by a feed-forward layer. In the Transformer, an input embedding layer converts the input sensor measurements to vectors of dimension d_{model}. The attention layer contains one of the most widely adopted attention modules, i.e., the Scaled Dot-Product Attention module, shown in Figure 3.4 (right). This module performs a matrix multiplication first. The outcome of the dot-product is then scaled by a constant factor $1/d_{model}$[9]. The attention modules output a weighted sum of the values that are denoted by V in Figure 3.4. Its weights are calculated as the softmax output w.r.t the dot-product which measures the similarity of each query (Q)

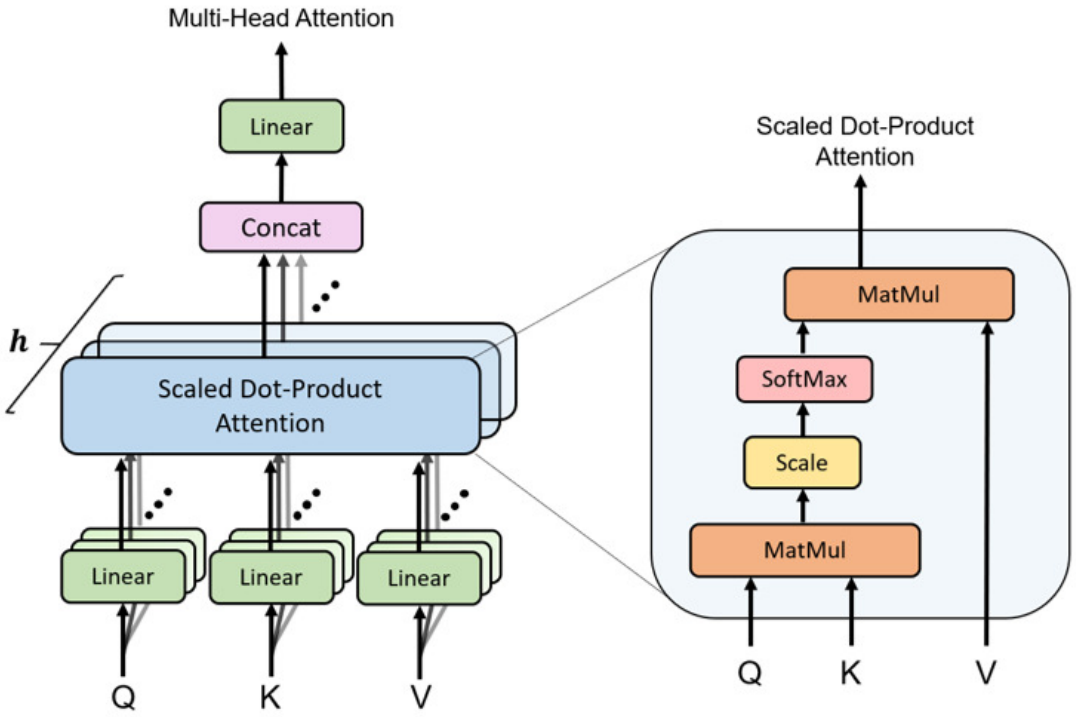


Figure 3.4: Multi-Head Attention based on parallel layers of Scaled Dot-Product Attention.

to the keys (K).

Similar to the use of multiple feature maps in CNNs, Transformers take advantage of multi-head attention, by making use of h independent heads. To realize this, h pairs of different linear projections transform d_{model}-dimensional queries, keys, and values to d_k, d_k, and d_v dimensions, respectively. Each of those projected vectors proceeds to the designated head, as shown in Figure 3.4. The heads apply the attention function in parallel. The following concatenation layer concatenates the outputs of the h heads, and those are projected to d_{model} dimensions with the last linear projection layer. The output of the attention proceeds to a FFN that comprises two linear transformations. The first layer linearly transforms from d_{model} to d_{ff} dimensions, and this inner layer uses the rectified linear unit (ReLU) as the activation function. Then, the following transformation produces the output

of the FFN. The sub-layers explained above, i.e., the multi-head attention and the FFN, are the building blocks for both the encoder and the decoder.

Different from vanilla Transformers[9], the Transformer network considered in this work contains two types of encoder: a sensor encoder, and a timestep encoder. The former is deployed to weigh different sensors by self-attention, while the latter serves to extract the feature from different timesteps. Each type of encoder is structured by stacking N_{enc} identical layers, and the two encoders work in parallel. The following feature fusion layer concatenates the two representations generated by the parallel encoders. The decoder is constructed by piling up N_{dec} layers, each one composed of two attention blocks followed by one FNN. Similar to the encoder, the attention block at the beginning takes the embeddings w.r.t. the decoder input and computes their values. The next attention block receives the output of the previous multi-head attention block, as well as the output of the feature fusion layer, to look back at what the encoder input sequences were. Finally, the output layer at the very end of the Transformer converts the decoder output to the predicted RUL.

Table 3.2: Architecture parameters of the Transformer and their bounds.

Parameter	Description	Min	Max
d_{model}	dim. of embedding and each sub-layer input/output	6	25
d_k	dim. of attention key	6	25
d_v	dim. of attention value	6	25
d_{ff_s}	dim. of FFN in sensor encoder layer	6	25
d_{ff_t}	dim. of FFN in time encoder layer	6	25
d_{ff_d}	dim. of FFN in decoder layer	6	25
h_s	number of attention heads in sensor encoder layer	1	16
h_t	number of attention heads in time encoder layer	1	16
h_d	number of attention heads in decoder layer	1	16
N_{enc}	number of encoder layers stacked	1	3
N_{dec}	number of decoder layers stacked	1	3

Table 3.2 presents the 11 parameters that configure the Transformer architecture. The first six parameters are related to the dimensions of the representations; among these, the first three parameters determine the vector dimensions in one sub-layer, i.e., the multi-head attention, while the remaining three parameters determine the inner dimensions in another sub-layer, the FFN. To reduce the obtainable number of combinations within the span of possible dimensions, each dimension parameter is divided by a fixed value of 4 when we define the range of that parameter. The lower bound of the range is mainly based on d_{model}, since the smallest possible d_{model} should be larger than the upper bound of h, 16. The specific range of dimensions is determined empirically. In particular, based on preliminary observations, we have found that a d_{model} of 16 cannot decrease the training loss and for this reason, d_{model} should be larger than 24. From this lower bound $6 = 24/4$, the upper bound is set to $25 = 100/4$, so that we can avoid combinatorial explosion by limiting the range of the parameter to 20 integer values. We consider the same range for all six parameters regarding the dimension.

The next three parameters indicate the number of attention heads. Its upper bound, 16, is inspired by the existing work[76], which examines the correlation between the number of heads and performance.

Finally, the last two parameters indicate the number of identical layers stacked when we generate the encoder and decoder. The maximum allowed number of stacks is set to 3. With these upper bounds, the largest network fits the available memory used in our work.

Overall, the search space defined by the above 11 parameters includes $2.36 \times 10^{12} \approx 20^6 \cdot 16^3 \cdot 3^2$ Transformer configurations.

3.2 Fitness evaluation

Our NAS algorithm involves the validation loss evaluation of the networks following their full training. When the given data are made up of a training set D_{train} and a test set D_{test}, D_{train} is split into D_w and D_v, where the former is a set of training purpose data and the latter is used for validation purpose: specifically, randomly chosen 80% of the engines in each sub-dataset are assigned to D_w which is used to produce training samples for solving Equation (3.4). The remaining 20% are designated as D_v that is used to evaluate the validation loss when we solve Equation (3.3). This proportion has been determined according to the investigation conducted in our recent study[12].

NNs trained by means of iterative gradient-based computations are generally referred to as back propagation-neural networks (BPNNs). In the case of BPNNs' architecture optimization, we can formulate the task described above as follows. Let T and S indicate the length of a time window and the number of sensors respectively. We consider an input sequence data X determined by the window sliding over the multi-sensor measurements. In other words, the input sequence comprises the measurements of the S sensors over T timesteps, and each position in the sequence corresponds to each timestep. This sequence can be written as $X = [x_1, \ldots, x_T]^\top$ with $x_t \in \mathbb{R}^{\mathrm{S}}$.

Considering a mini-batch of bs training samples, the loss function $l(\cdot)$ of the network corresponds to the Mean Squared Error (MSE) between the output of the network, $Y = [y_1, \ldots, y_{bs}]$ and its ground truth labels, $Z = [z_1, \ldots, z_{bs}]$:

$$l\left(Y, Z\right) = \frac{1}{bs} \sum_{j=1}^{bs} \left(y_j - z_j\right)^2 \tag{3.1}$$

where y_j denotes the predicted RUL w.r.t. X_j while z_j represents the ground truth RUL for the j-th sample X_j. Note that in this work we always take the

RUL value at the last timestep T of the sequence X.

An exception is a MOO task focused on optimizing the ELM (Section 4.1.1) which receives a single timestep input rather than the sequential input by time window. We use the same setup for defining D_w and D_v, while the training does not rely on the BP algorithm but its own algorithm explained in Section 4.1.1.

3.3 Fitness prediction

NAS consists in solving an optimization problem that is formally written in:

$$a^* = \arg\min_a f(a) \tag{3.2}$$

where a denotes a given architecture from the search space, and a^* indicates the optimal architecture regarding the objective function f.

The following equations define f:

$$f(a) = \mathcal{L}_{val}(w^*(a), a) \tag{3.3}$$

$$w^*(a) = \arg\min_w \mathcal{L}_{train}(w, a) \tag{3.4}$$

For a given architecture a, the function value, $f(a)$, is an observation of the network performance with the trained weights $w^*(a)$, where the performance corresponds to the validation loss $\mathcal{L}_{val}$. The training phase, described by Equation (3.4), indicates that $w^*(a)$ can be obtained by fully training the network for a given architecture a. Training by back-propagation requires iterative gradient computation, which is computationally expensive. Thus, it is necessary to adapt performance prediction strategies for avoiding this high computational cost.

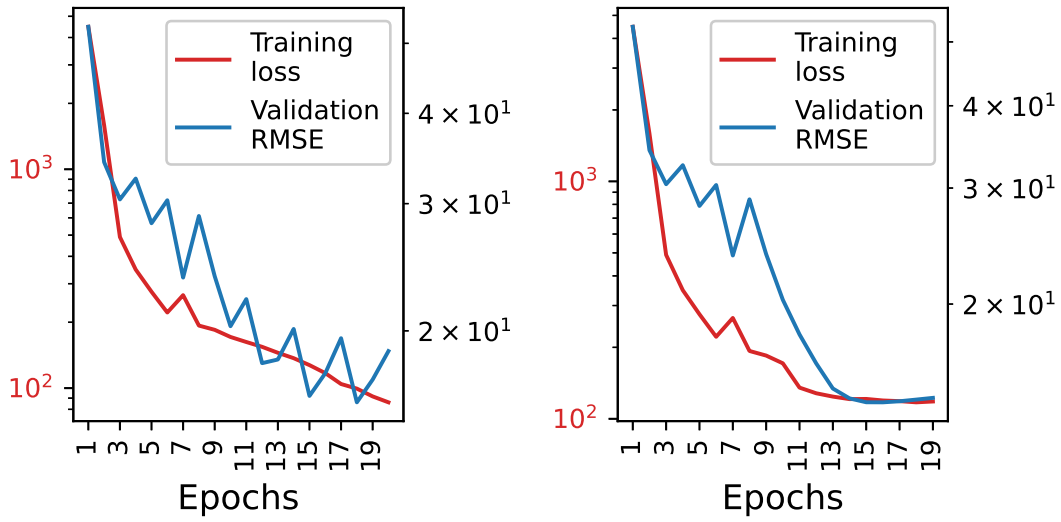
In the following, the first two strategies introduced in Sections 3.3.1 and 3.3.2 offer to save the number of epochs when we train each network by Equation (3.4); while zero-cost proxy shown in Section 3.3.3 make it possible

to eschew the computationally intensive network training; lastly, we present a model-based performance predictor which employs a regression model as a surrogate for Equation (3.3) denoting the performance observation of the trained network.

3.3.1 Early-stopping

If all the networks to be evaluated during the evolutionary search are trained with the same fixed number of epochs, the training may include unnecessary training efforts which do not improve the validation loss. In order to reduce computing time by discarding the redundant training epochs, we present an early-stopping strategy in conjunction with learning-rate decay. It reduces the learning-rate by a constant multiplicative factor called gamma when the training epoch arrives at one of the predefined epochs reserved for decay. Early-stopping monitors the validation loss every epoch and stops the training when no improvement is observed within a number of epochs. This early-stopping strategy is advantageous for reducing training epochs as well as avoiding overfitting.

For instance, Figure 3.5 presents the learning curve of a multi-head CNN-LSTM with and without applying learning-rate decay. In detail, the learning-rate for Figure 3.5 (a) remains at its initial value of 10^{-3} across all epochs, whereas the curve shown in Figure 3.5 (b) highlights the effect of learning-rate decay with a gamma value of 0.1. In the former, the training lasts until the maximum number of epochs of 20 even if early-stopping is applied; the learning-rate remains high in all epochs, and the validation curve keeps fluctuating. Because of this, the training keeps undesirably finding lower validation loss within 5 epochs of patience. In contrast, we multiply the learning-rate by 0.1 at epochs 10 and 15. With this learning-rate decay, the validation curve in Figure 3.5 (b) reaches the lowest value at epoch 14, and the training is stopped at epoch 19.



(a) (b)

Figure 3.5: 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.

Thanks to the early-stopping strategy in conjunction with the decay, we are able to set an initial learning-rate value to a relatively large value without being concerned about the fluctuation which disturbs the proper evaluation of each individual for comparison with others. Thus, in essence, this strategy is needed and taken into account when we use Equation (3.4) for a reliable fitness evaluation by Equation (3.3).

3.3.2 Learning curve extrapolation

Extrapolation of learning curves is a fitness prediction strategy that allows save a predefined number of training epochs by fitting and extrapolating the validation loss curve. Although the early-stopping discussed in Section 3.3.1 helps save a few training epochs , the benefit varies depending on how we define no improvement on the validation loss and the degree of patience. The validation curve extrapolation strategy described below can alleviate this problem and always endow the same benefits.

As shown in Figure 3.5 (b), proper scheduling of the learning-rate changes of the optimizer leads to the trend of the validation loss change being similar for all the networks in our search space when they are trained. Taking this into account, we formulate the learning curves (e.g., the validation RMSE curves in our work) based on a set of functions $f(x)$ where the shape of each of them coincides with the trend of the validation RMSE. In particular, we chose the k functions ($k = 5$ in this work) specified in Table 3.3 from the literature[77] and set the values of the parameters (that are denoted by α, β, γ, and δ) in the following way: we train the network for n_t epochs observing the validation RMSE, using those observations is fitting each function based on non-linear least squares minimization, which is defined by Section 3.3.2:

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

where y_j^o indicates the observed validation RMSE at x_j, while $f(x_j)$ denotes the function value at x_j. The Levenberg-Marquardt algorithm[78] is used to solve the above least squares problem, and the function derived by curve fitting is denoted as f^*.

Table 3.3: Functions $f(x)$ used for extrapolation of learning curves.

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	$\alpha + \frac{\beta-\alpha}{1+10^{(x-\gamma)\delta}}$

Figure 3.6 visualizes the shape of the obtained function f^*, and we can observe that no single function can fully delineate the validation RMSE curve. To obtain a function closer to our learning curve, we combine all the obtained

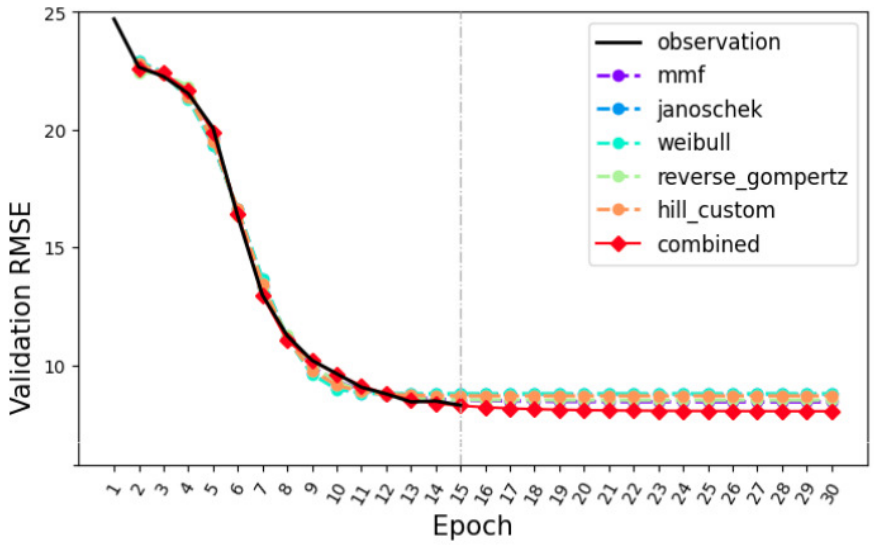


Figure 3.6: An example of how the learning curve is derived from the $k = 5$ functions and the observations for $n_t = 15$ epochs.

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 different f^* for n_t epochs, and $\boldsymbol{y}^o \in \mathbb{R}^{n_t}$ is a vector of observations also for n_t epochs. The optimal $\boldsymbol{a} \in \mathbb{R}^k$, obtained by solving the linear problem, can be written as $\boldsymbol{a}^* = [a_1^*, \cdots, a_k^*]$.

For instance, Figure 3.6 describes how the learning curve extrapolation works when we train a network for $n_t = 15$ epochs and take the predicted value from the red-colored curve at $n_m = 30$ epoch. Specifically, we collect the function value of each f^* at x_{n_m} denoted as $\boldsymbol{f}^*(x_{n_m}) = [f_1^*(x_{n_m}), \cdots, f_k^*(x_{n_m})]$. The target value $y_{n_m}^p$ is then calculated by the linear combination of those values where the weights are $\boldsymbol{a}^*$. Hence, this is written as follows:

$$y_{n_m}^p = \boldsymbol{f}^*(x_{n_m}) \cdot \boldsymbol{a}^*. \tag{3.5}$$

If the validation RMSE has not fully converged at n_t epochs, then our defined

curve can keep decreasing with x. This indicates that the minimum observed value within n_t epochs, $\min(\boldsymbol{y}^o)$, is greater than the value calculated at n_t epoch, $y^p_{n_m}$; the difference between them is defined as $d = \min(\boldsymbol{y}^o) - y^p_{n_m}$ so that we assign different fitness value based on the sign of d (positive or negative). In particular, the predicted fitness is defined as follows:

$$fitness_{RMSE} = \begin{cases} y^p_{n_m}, & d > 0 \\ \min(\boldsymbol{y}^o) - |d|, & d \leq 0. \end{cases} \tag{3.6}$$

If d is positive, then $y^p_{n_m}$ is directly used as the fitness value. Otherwise, the fitness value is determined as $\min(\boldsymbol{y}^o) - |d|$.

The learning curve of each network considered during the search process starts converging at a different epoch. In particular, the learning curve of some networks reaches a plateau in the first few epochs and does not decrease with x, i.e., d can be less than or equal to 0. However, the actual validation loss may decrease with x even a little, since the network continues to be trained every epoch. In this scenario, we subtract the absolute value of d from the minimum observed value $\min(\boldsymbol{y}^o)$ and use $min(\boldsymbol{y}^o) - |d|$ as the predicted fitness value instead of directly taking $y^p_{n_m}$. As such, we allocate a lower fitness value to the network even if its learning curve converges to a high value in the first few epochs and does not decrease with x.

3.3.3 Zero-cost proxy

One of the zero-cost proxies[79], a metric called architecture score without training[80], can be considered as a fitness prediction strategy in our NAS process.

Each DL model is trained by tuning its parameters according to Equation (3.4), and this is still computationally demanding even if we cut several redundant training epochs. On the contrary, the architecture score without training is a proxy that represents the final performance at initialization (i.e.,

right after initializing a network but before its training), where the final performance here corresponds to the evaluated fitness (i.e., validation loss in our work) after the training. This training-free proxy quantifies how well the network at initialization discriminates the different inputs.

Given that rectified linear unit (ReLU) is commonly used as the activation function in the networks, the output activation of each unit can be an indicator the unit is active or not; if the activation output has a non-zero positive value, then the unit is active; otherwise, it is inactive. In the former case, we set the activation output value to 1 for the former, while it is set to 0 for the latter. I.e., the output is encoded as a bit.

In detail, when a network containing N_{ReLU} activation units is fed by an input sample x_i, we obtain a binary code $\boldsymbol{c}_i \in \{0,1\}^{N_{ReLU}}$ by gathering all the bits from each unit. In this way, a mini-batch $\mathrm{X} = \{x_i\}_{i=1}^{M}$ yields M binary codes. Here, the similarity of two binary codes from two different inputs reveals how difficult it is for the network to separate them. For instance, suppose there are two different inputs that are particularly difficult to distinguish. If one network produces the same binary code while the other network gives us different codes, then we consider the latter to be a better network than the former in terms of performance which is related to the ability to discriminate between similar inputs.

Let x_i and x_j be two different inputs within a mini-batch X, and the two binary codes for these inputs be written as $\boldsymbol{c}_i$ and $\boldsymbol{c}_j$ respectively. The similarity between them is then measured by the Hamming distance $d_H(\boldsymbol{c}_i, \boldsymbol{c}_j)$. A proxy for the fitness evaluation is calculated by the following equations:

$$\boldsymbol{K}_H = \begin{bmatrix} N_{ReLU} - d_H(\boldsymbol{c}_1, \boldsymbol{c}_1) & \cdots & N_{ReLU} - d_H(\boldsymbol{c}_1, \boldsymbol{c}_M) \\ \vdots & \ddots & \vdots \\ N_{ReLU} - d_H(\boldsymbol{c}_M, \boldsymbol{c}_1) & \cdots & N_{ReLU} - d_H(\boldsymbol{c}_M, \boldsymbol{c}_M) \end{bmatrix}, \tag{3.7}$$

$$s = \frac{c}{\ln |\boldsymbol{K}_H|} \tag{3.8}$$

where $\boldsymbol{K}_H$ denotes the kernel matrix that computes the correspondence between binary codes for X, and s represents the proxy metric. Following Equation (3.8), the determinant of the kernel matrix $|\boldsymbol{K}_H|$ is higher as the kernel approximates a diagonal matrix, and large distances between two different codes mean that those can be separated well by the network. Thus, a lower value of the proxy for the same input batch at initialization implies lower validation RMSE after training.

3.3.4 Model-based performance predictor

One possible performance estimation strategy is to consider the optimization problem shown in Equation (3.2) as a supervised ML problem and solve it by employing a regression model. More specifically, we prepare a regression model before solving the optimization. Here, the preparation is also referred to as the initialization step[81], and requires: 1) collecting a fixed number m of pairs $(a, f(a))$; and 2) fitting a regression model based upon this collection. The trained regression model $\hat{f}$ can provide an approximation of f. Thus, the regression model serves as a surrogate for the validation loss observation, i.e., we apply a surrogate model $\hat{f}$ which can approximate f.

When it comes to supervised learning, it is necessary to collect labeled training samples. Since such a process requires the performance evaluation of fully trained networks, the amount of labels is typically very restricted. Nevertheless, the model trained with limited data should be able to predict the performance of individual networks all over the parameter space.

To this end, we choose NGBoost[82] as the surrogate model $\hat{f}$. Different from typical regression models that return a single best guess prediction (namely, a point estimation), NGBoost allows for predictive uncertainty estimation and outputs a full probability distribution. To be more specific, NGBoost is a modular algorithm that is composed of three components, as shown Figure 3.7. The algorithm generalizes gradient boosting[83] and makes

use of natural gradients and boosting to integrate three modular components: Base Learners, Parametric Probability Distribution, and Scoring Rules. In each iteration of the learning algorithm, a vector representation of the base learner's parameters (θ) for the current model input X is fed into the Distribution component, which determines the probability distribution P_θ. In the following Scoring Rules component, the scoring function S is defined based on the distribution P_θ and the prediction target y. Finally, the natural gradient of S w.r.t. θ is used to fit the Base Learners.

We select the decision tree as Base Learners l, while the conditional probability in the second component follows the Normal distribution. The logarithmic scoring rule[82] is considered as S. Our choice of the NGBoost components and hyper-parameter tuning on the model follow the settings used for all experiments reported in[82].

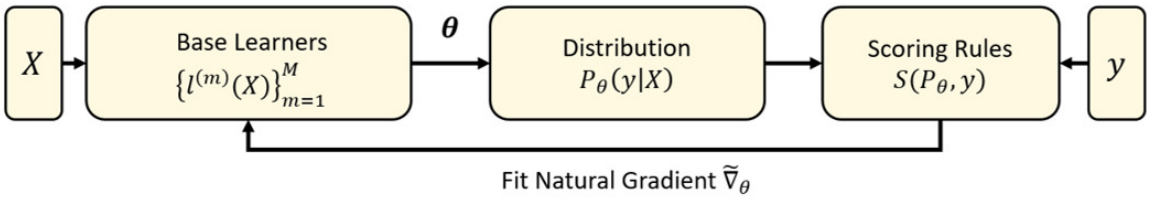


Figure 3.7: Overview of NGBoost which comprises three modular components: Base Learners (l), Parametric Probability Distribution (P_θ), and Scoring Rule (S).

In addition, the following two crucial aspects should be considered in use the NGBoost as a surrogate model in our NAS process. Because the full training of the network is computationally very expensive, we can prepare very few samples to train the predictor described above. On the other hand, the parameter space defined in Section 3.1 is extremely large compared to the feasible number of the prepared samples. Therefore, we need to acquire the maximum information with the minimum number of samples, by spreading them out with the aim of encouraging a diversity of data. In other words, it

is necessary to handle the matter of choosing sample points in the parameter space to train the surrogate, so that it has a good space-filling property. To achieve this, we apply Latin hypercube sampling (LHS)[84] when we prepare the training samples for our performance prediction model. In this sampling strategy, each sample "remembers" in which part of the search space it was taken, so that each space dimension is evenly sampled.

As another effort to improve the surrogate model, two additional parameters regarding the network are added to the vector described in Section 3.1 when we prepare the input for the surrogate model. More specifically, we concatenate the integer values for the architecture parameters of the given network with the following two values: the number of trainable parameters in the network, and the single-shot network pruning (SNIP) value[85]. The SNIP value is a pruning at initialization metric that computes a saliency metric at initialization, and it can approximate the change in the loss at initialization w.r.t. removing a specific connection. This metric was originally proposed to find sparse networks, but it has been used also for estimating the performance of lightweight networks, considering the correlation between the SNIP value and the performance at initialization[81]. In order to let the model consider additional information about the performance of a given network, we feed the SNIP value to the regression model as an additional input.

3.4 Predictor-assisted evolutionary NAS algorithms

In this section, we introduce two evolutionary NAS algorithms for developing RUL prediction models. The first one is dubbed as the "evolutionary NAS for predictive maintenance (ENAS-PdM)[12]". It uses a GA to explore the combinatorial parameter space of the multi-head CNN-LSTM (Section 3.1.1) so as to find the best architectures for predictive maintenance which is achieved by solving the RUL prediction task. The early-stopping described in Sec-

tion 3.3.1 is used as a fitness prediction strategy. To reduce the computational cost of the evolutionary search, we also implement a history mechanism that helps avoid the redundant fitness evaluation of individuals previously evaluated. In the second algorithm, referred to as the "surrogate-assisted ENAS algorithm[13]", we further improve the ENAS-PdM algorithm by incorporating a surrogate into the GA; a model-based performance predictor described in Section 3.3.4 is considered as a surrogate model. We use this algorithm to discover the optimal architectures of the Transformer detailed in Section 3.1.2.

The pseudo-code of the proposed ENAS-PdM algorithm is shown in Algorithm 1. It starts by initializing the population; where n_{pop} indicates the population size, we generate $n_{pop} - 1$ individuals at random and take the parameters of a well-performing human-designed architecture[68] as the remaining individual. This super-fit mechanism[86] enables our GA to start with a good individual by including it in the initial population.

The fitness evaluation follows the procedure specified in Section 3.2; for each individual, we construct a multi-head CNN-LSTM network based on a vector containing the architecture's parameters, and the fitness of the network (the validation RMSE in our work) is evaluated after training it. In the procedure, when an individual first appears in the search process, then its fitness is evaluated and recorded in a history table. Otherwise, we get the fitness value from the table instead of evaluating it again.

As for genetic operators in our GA, mutation and crossover are used to ensure a good trade-off between exploration and extrapolation. We apply each of them independently with a probability of 0.5, i.e., $p_{mut} = p_{cx} = 0.5$ where p_{mut} and p_{cx} denote mutation probability and crossover probability respectively. In this way, individuals are produced by either mutation or crossover (exclusively), and this allows the algorithm to avoid generating disruptive combinations of mutation and crossover that could lead to bad

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

```
1:  function EVOLUTION(a, b)
2:      pop ← initialize_pop()
3:      evaluated ← Set()                                  ▷ Evaluated individuals
4:      for (gen = 0; gen < generations; gen+ = 1) do
5:          evaluate_fitness(evaluated, pop)
6:          new_pop ← select(pop)
7:          new_pop ← crossover(new_pop)
8:          new_pop ← mutation(new_pop)
9:          pop ← check_parents(pop, new_pop)
10:     end for
11:     return best(pop)
12: end function
13:
14: procedure EVALUATE_FITNESS(evaluated, pop)
15:     for ind ∈ pop do
16:         if ind ∉ evaluated then
17:             ind.fitness ← fitness(ind)
18:             evaluated.add(ind)
19:         else
20:             ind.fitness ← evaluated.get(ind).fitness
21:         end if
22:     end for
23: end procedure
```

individuals. Regarding crossover, we use a specialized one-point crossover that first ranks the individuals by their fitness, 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 operator allows us to *exploit* the best individuals, trying to combine them with 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. The common opinion about EAs is that exploration should be the task of mutation, while we do not limit exploration to the of mutation. Based on discussions on evolutionary exploration and exploitation [87,88] questioning the common belief, we consider exploration in the above strategy.

We then apply uniform mutation to the population (containing the offspring generated by crossover and individuals that did not undergo crossover), in which, according to a probability p_{gene}, each gene (i.e., one of the architecture parameters) can be mutated to another value uniformly

drawn from its bounds described in Table 3.1. The p_{gene} is set to 0.3 so that the expected number of mutations is set between 1 and 2; 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 of 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).

In the following, we present the surrogate-assisted ENAS algorithm that is developed for finding the optimal architectures of the Transformer described in Section 3.1.2. It uses the surrogate model explained in Section 3.3.4 to further reduce the computational burden associated with the fitness evaluation of the evolutionary search algorithm.

As shown in Algorithm 2, it starts with n_{pop} randomly generated individuals. Each run of the algorithm terminates when it reaches a predetermined maximum number of generations n_{gen}, similar to Algorithm 1. As we can see in Figure 3.8, the phenotype representation of each individual is the Transformer architecture, a, associated with its genotype which is an integer vector containing the parameters described in Table 3.2.

The model-based performance predictor defined in Section 3.3.4 is used as the surrogate model $\hat{f}$, which should be initialized before the algorithm starts its main loop. Here, the predictor is the probabilistic regression model, and we train it according to the following procedures. First, we sample a fixed number m of architectures from the search space, by means of LHS. We then prepare the labeled training samples that consist of m pairs $(a, f(a))$ where $f(a)$ indicates the validation RMSE of the selected a after its full training. Lastly, the collected pairs are stored in memory to avoid fully training the

same architecture a again. Besides, every collected pair during the main loop of the search algorithm is stored as well for the same reason.

Algorithm 2 Pseudocode of the surrogate-assisted ENAS algorithm.

```
1:  function EVOLUTION(n_pop, n_gen)
2:      pop ← initialize_pop(n_pop)                          ▷ n_pop: population size
3:      f̂(·) ← initialize_predictor()                        ▷ probabilistic regression model (NGBoost)
4:      history ← Set()                                      ▷ evaluated individuals by full training
5:      for (gen = 0; gen < n_gen; gen++) do                 ▷ n_gen: max. number of generations
6:          evaluate_fitness(pop, f̂(ind), history)
7:          new_pop ← select(pop)
8:          new_pop ← crossover(new_pop)
9:          new_pop ← mutation(new_pop)
10:         pop ← check_parents(pop, new_pop)
11:     end for
12:     return history
13: end function
14:
15: procedure EVALUATE_FITNESS(pop, f̂(·), history)
16:     for ind ∈ pop do
17:         ind.fitness ← f̂(ind)
18:     end for
19:     topk ← reordering(pop)        ▷ reorder individuals w.r.t. predicted fitness and select top k individuals
20:     for ind ∈ topk do
21:         if ind ∉ history then
22:             a ← phenotype_decoding(ind)
23:             ind.fitness ← f(a)                           ▷ full training to evaluate the fitness
24:             history.add(ind)
25:         else
26:             ind.fitness ← history.get(ind).fitness
27:         end if
28:     end for
29:     retraining(f̂(·), topk)                               ▷ update the predictor with the new observations
30: end procedure
```

The main difference between Algorithm 1[12] and Algorithm 2[13] is the fitness evaluation procedure. As depicted in Figure 3.8, the fitness evaluation leveraging the surrogate model is done in three steps: fitness prediction, reordering, and fitness observation. In the first step, the surrogate model $\hat{f}$ produces the fitness of the individuals. We can see the main advantage of the proposed algorithm here; making a prediction by the surrogate has almost no cost compared to the full training for the actual fitness evaluation. We need to spend a computational budget to initialize the predictor, but this enables us to avoid full training on many networks whose fitness needs to be

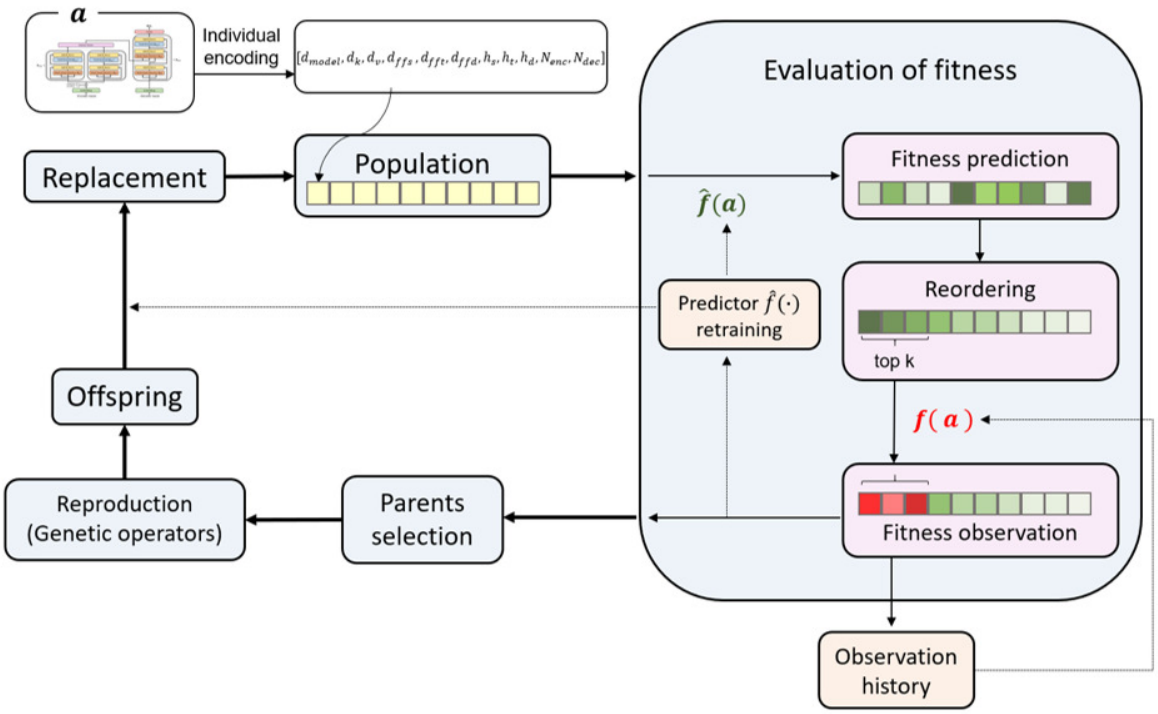


Figure 3.8: Overview of the surrogate-assisted evolutionary algorithm for NAS.

evaluated during the evolutionary process.

After marking each individual with the predicted fitness, we sort them according to their predicted fitness. The ranking by the predictions can be used as it is in the evolutionary process because the predictions given by the regression model broadly correlate with the actual fitness observations. However, the correlation may not be very high, since the predictor is trained on a limited number of samples evenly spread over a large search space. For this reason, we propose to perform fitness observation (which indicates fitness evaluation by means of full training) on the *elites*, i.e., the top k individuals based on the predicted fitness, that are expected to have also better actual fitness values. For the current elites that are made up of individuals that have good fitness, we replace the predictions with the observations. By doing so, we can improve the correlation between the fitness predictions and the fitness

observations, at least for the individuals that have good fitness.

Furthermore, in each generation, the predictor is updated according to the new observations; the obtained $(a, f(a))$ pairs are appended to the training data for the predictor, and the predictor is retrained. In this way, the knowledge obtained by the new observations improves to fitness predictions. At the same time, those pairs are recorded in the history, which is a look-up table used to avoid redundant computation (as mentioned above). In fact, before carrying out a fitness observation on a, first we check if a exists in the history. Then, we take its fitness $f(a)$ from the history if it has already been evaluated, otherwise, we perform the fitness observation for a and add the observation to the history.

After evaluating fitness, every individual in the current population is considered to induce offspring. Specifically, the genetic operators considered in our work are crossover and mutation. Each of them is applied independently, to avoid disruptive combinations of their effects. First of all, reproduction starts with a custom one-point crossover that, with a probability of 0.5, mates two adjacent individuals, i.e., given that the individuals are sorted according to fitness, the $(2i-1)$-th best individual is combined with the $(2i)$-th best individual, where $i \in [1, \frac{n_{pop}}{2}]$. The offspring population is then obtained by applying, again with a probability of 0.5, uniform mutation to the population that includes both the individuals obtained through crossover and the individuals that have not undergone crossover. During mutation, according to a probability p_{gene}, we replace the value of each gene with a uniform random value drawn between its upper and lower bounds. The value of p_{gene} is set to 0.3, so that it can lead to mutating an average of 3.3 genes (out of 11) per individual. This way, we can achieve a good compromise between excessively small or excessively disruptive mutations.

Finally, the population for the next generation is formed by the following replacement: 1) we predict the fitness of each offspring; 2) the fitness of its

parents is checked; 3) if the offspring's fitness is superior compared to the fitness of one (or both) of its parents, that parent (or the one with the worst fitness, if both parents are worse than the offspring) is replaced with the offspring.

Chapter 4

Multi-objective optimization of neural architectures

Whereas Chapter 3 pursues the development of RUL prediction networks seeking lower prediction error, our works discussed in this chapter use the evolutionary NAS to achieve a trade-off between prediction error and the number of trainable parameters. Since the motivation of those works is to develop a proper RUL prediction tool to cope with constrained HW resources, we consider the MOO of relatively simple networks that have primitive structures such as ELM, CELM, and 1-D CNN, compared to the very deep networks discussed in the previous chapter.

For the works discussed in this chapter, we apply NSGA-II to search for the best solutions. The first work is based on Mo et al.[14] that proposes to optimize the ELM architecture which allows extremely fast training without iterative gradient computation and this, in turn, does not require performance prediction. We then describe an extension of the first work; the extension is based on Mo et al.[15] that applies the MOO algorithm to optimize the architecture of the CELM. The next work that is based on the paper by Mo et al.[16] proposes to optimize the architecture of the 1-D CNN and to accelerate the search by leveraging the two performance prediction techniques: zero-cost proxy (Section 3.3.3) and learning curve extrapolation

(Section 3.3.2).

4.1 Individual encoding

In this section, we describe the baseline structure of the ELM, CELM, and 1-D CNN, with details on the architecture parameters we optimize.

4.1.1 Extreme learning machine

An ELM is a novel and fast training algorithm for *single-hidden layer feed-forward neural networks* (SLFNs). In essence, the training of NNs corresponds to finding the optimal weights as defined by Equation (3.4), and the gradient-based optimization method (in particular, the back-propagation algorithm) is widely used to tune weights of the networks; the training of BPNNs is usually time-consuming and computationally burdensome. In contrast, an ELM is able to tune the weights of SLFNs in a very fast and efficient way; it randomly initializes the input weights and merely considers an analytically determined non-iterative solution as the output weights.

The seminal paper on ELMs[10] claims that they are not only very fast in training but also competitive in performance compared to BPNNs[1]. On various regression tasks, ELMs have shown comparable performance in terms of prediction accuracy[89,90] and generalization[91,92]. Particularly, the existing work[89] reports the performance of the handcrafted ELMs on the RUL prediction task and highlights that employing the ELMs is advantageous for saving training time, while their performance is comparable to BPNNs in terms of RUL prediction error. Considering the advantages of ELMs and their promising results in the previous works, here we optimize an ELM to use it as an RUL prediction tool providing both good prediction accuracy

[1]We should note that, strictly speaking, "ELM" refers to the training algorithm only. However in this work "ELM" is generically used to refer to both the training algorithm and the neural network itself.

and short learning time given by their lower number of trainable parameters, compared to BPNNs.

As for training a SLFN comprising L hidden neurons by N labeled training samples, an ELM can be formally described as follows. Each training sample is made up of a d-dimensional input vector with a corresponding c-dimensional label, and an input and its label are denoted by $\boldsymbol{x}_i$ and $\boldsymbol{t}_i$ respectively. A given set of N training samples can then be written as $(\boldsymbol{x}_i, \boldsymbol{t}_i)$, $i \in [1, N]$ with $\boldsymbol{x}_i \in \mathbb{R}^{\mathrm{d}}$ and $\boldsymbol{t}_i \in \mathbb{R}^{\mathrm{c}}$. When it comes to a data-driven RUL prediction model, d is the number of monitoring signals, and $c = 1$ (i.e., the label is a scalar real number representing a RUL value).

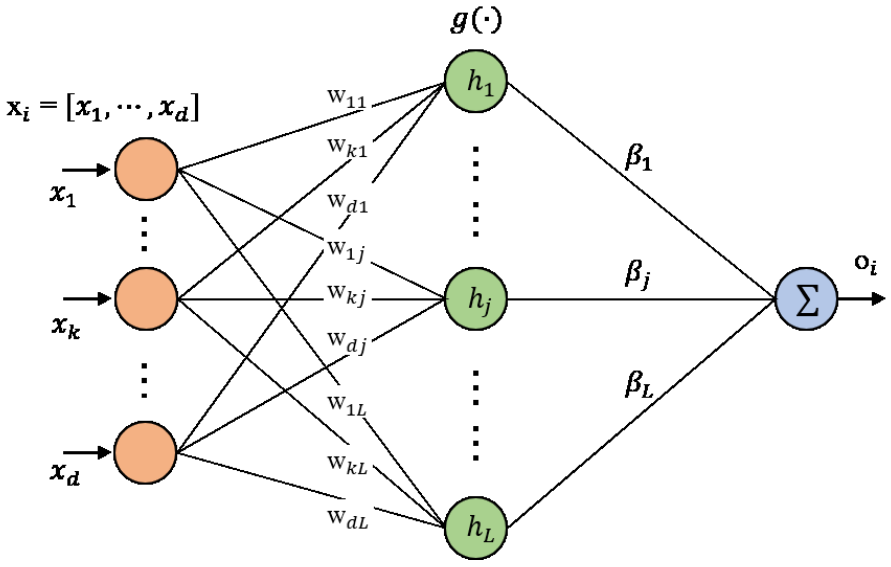


Figure 4.1: Illustration of ELM with the structure of a SLFN.

Figure 4.1 details the structure of a SLFN and the notation used for the mathematical expression of the ELM. The network depicted in the figure consists of three parts: d input nodes, a single hidden layer of L neurons with activation function $g(\cdot)$, and a single output node. For a given input sample $\boldsymbol{x}_i$, the output of the network o_i with L hidden neurons and activation

function $g(\cdot)$ is defined by:

$$\mathrm{o}_i = \sum_{j=1}^{L} \beta_j g(\boldsymbol{w}_j \cdot \boldsymbol{x}_i + b_j). \tag{4.1}$$

where $\boldsymbol{w}_j = [w_{1j}, \ldots, w_{dj}]$ is a vector of weights on the connections between the d input neurons (which are assumed to be linear) and each j-th hidden neuron, β_j is the weight on the connection between the j-th hidden neuron and the output neuron, and b_j denotes the bias for the j-th hidden neuron. The computation for all the N equations (one for each of the N samples) can be written compactly as:

$$\boldsymbol{H} \cdot \boldsymbol{\beta} = \begin{bmatrix} g(\boldsymbol{w}_1 \cdot \boldsymbol{x}_1 + b_1) & \cdots & g(\boldsymbol{w}_L \cdot \boldsymbol{x}_1 + b_L) \\ \vdots & \ddots & \vdots \\ g(\boldsymbol{w}_1 \cdot \boldsymbol{x}_N + b_1) & \cdots & g(\boldsymbol{w}_L \cdot \boldsymbol{x}_N + b_L) \end{bmatrix} \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_L \end{bmatrix} \tag{4.2}$$

where $\boldsymbol{H}$ is the hidden layer output matrix (of size $N \times L$), and $\boldsymbol{\beta}$ (of size L) consists of the weights of all the connections between the hidden neurons and the output neuron.

To train the SLFN defined above (i.e., the ELM) is equivalent to finding a least square solution $\hat{\boldsymbol{\beta}}$ to the linear system $\boldsymbol{H} \cdot \boldsymbol{\beta} = \boldsymbol{T}$ where $\boldsymbol{T} = [\boldsymbol{t}_1, \cdots, \boldsymbol{t}_N]^\top$. Therefore, the mathematical formulation of the training procedure can be expressed as:

$$\|\boldsymbol{H}\hat{\boldsymbol{\beta}} - \boldsymbol{T}\| = \min_{\boldsymbol{\beta}} \|\boldsymbol{H}\boldsymbol{\beta} - \boldsymbol{T}\| \tag{4.3}$$

As discussed in[93], the smallest norm least squares solution of the above equation is determined by:

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{H}^\top \boldsymbol{H} + \alpha \boldsymbol{I})^{-1} \boldsymbol{H}^\top \boldsymbol{T}. \tag{4.4}$$

where $(\boldsymbol{H}^\top \boldsymbol{H})^{-1} \boldsymbol{H}^\top$ is the Moore-Penrose generalized inverse of the matrix $\boldsymbol{H}$, and we add an L2 regularization term $\alpha \boldsymbol{I}$ (with $\alpha \in \mathbb{R}$ being an arbitrarily small value) to the inverse term $\boldsymbol{H}^\top \boldsymbol{H}$ so that it does not become

singular. While the ELM merely requires solving Equation (4.4) once to determine the output weights, BPNNs need to tune all the weights in them iteratively over several epochs.

Equation (4.4) informs us that the computational complexity of the ELM is $\mathcal{O}(NL^2 + L^3)$, and it is determined by the size of the matrix $\boldsymbol{H}$. The complexity is cubic w.r.t. the number of hidden neurons L, but a large value for L does not always contribute to improving prediction performance and may lead to overfitting.

As such, finding the optimal value of L, as well as of the constant value for proper L2 regularization, is a crucial element for ELM performance in terms of both the RUL prediction error and the number of trainable parameters. However, this task is not easily achievable by manual design or empiric considerations. Driven by this motivation, we use a GA to discover optimized ELMs automatically from the search space defined by the integer parameters described in Table 4.1.

Table 4.1: Parameters of the ELM to be optimized and their bounds.

Parameter	Description	Min	Max
n_{tanh}	number of hidden neurons with hyperbolic tangent activation	1	200
n_{sigm}	number of hidden neurons with sigmoid activation	1	200
r	L2 regularization parameter	2	6

We encode the number of hidden neurons having two different activation functions as n_{tanh} and n_{sigm}, respectively, since it was found that using different activation functions for $g(\cdot)$ in preliminary experiments produces different results. The remaining parameter, r, refers to the order of magnitude of the L2 regularization parameter α explained above, i.e., $\alpha = 10^{-r}$.

The lower and upper bounds (chosen empirically) for each parameter in the ELMs are also presented in Table 4.1. For the first two parameters

regarding the number of hidden neurons, it should be noted that their bounds are multiplied by a fixed value of 10 when we generate an ELM instance (i.e., the phenotype) so that we use a discretization on the number of hidden nodes to reduce the search space yet allowing ELMs of up to $2,000$ $tanh$ hidden neurons and $2,000$ sigmoid hidden neurons. In other words, both n_{tanh} and n_{sigm} range in the interval $[1, 200]$, while the corresponding number of hidden neurons can range between 10 and $2,000$ with a step size of 10. These values have been chosen empirically. In particular, the maximum value for L (given by the sum of the two kinds of neurons) is $4,000$, and this upper bound is determined to limit the size of the hidden layer output matrix $\boldsymbol{H}$, which is $N \times L$, while keeping its calculation affordable during the evolutionary search. Because the whole range of integers between 1 and $2,000$ would be too large to explore, we divide it by 10, to decrease the number of possible combinations for those two parameters: by doing so, we reduce the number of possible combinations from 4×10^6 to 4×10^4. For the remaining parameter, r, it defines α of Equation (4.4), and the value of α should be relatively small to avoid affecting the ELM performance, while being enough to prevent the inverse term in Equation (4.4) from becoming singular; in this work, we find that such an upper and a lower bound for α are 10^{-2} and 10^{-6} respectively, and this, in turn, decides the bounds for r.

4.1.2 Convolutional extreme learning machine

Figure 4.2 depicts a CNN structure for a regression task, such as the RUL prediction, on multivariate time series. The network mainly comprises the feature extraction stage and the regression stage. The feature extraction stage consists of a set of 1-D convolutional layers aiming to extract high-level feature representations, while the following regression stage is a fully-connected layer computing an output RUL value from the extracted features.

BP algorithms are extensively used to train CNNs, but gradient-based

learning algorithms are in general slower than required because they tune all the parameters of the network iteratively. Considering that the slow speed of BP algorithms can be a major bottleneck in the applications of CNNs, here we apply CELMs, which are fast training as they do not require an iterative gradient computation. Similar to the ELM, described in Section 4.1.1, the training of CELMs consists in analytically determining the output weights on the connections between the hidden neurons of the fully-connected layer and the output neuron, in conjunction with a random initialization of all the remaining weights. In other words, the convolutional filters are randomly generated, and we randomly choose the input weights to the fully-connected layer as well.

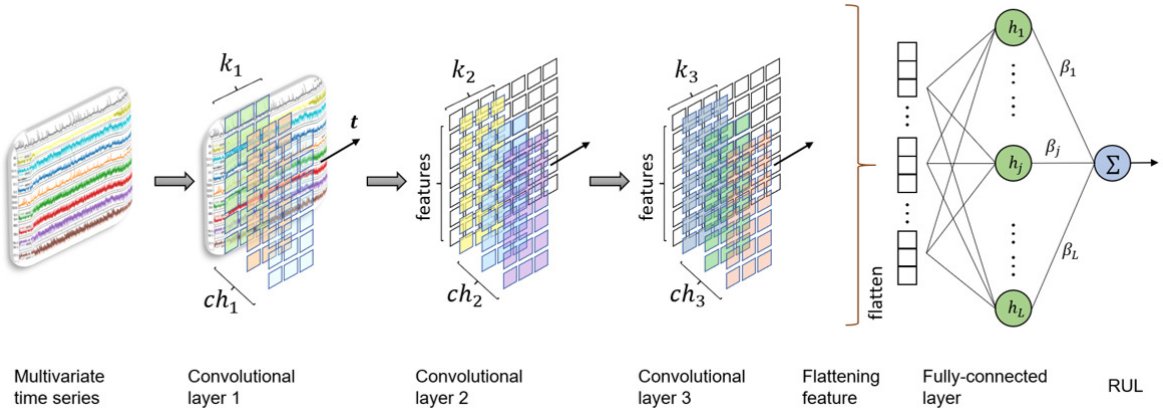


Figure 4.2: Illustration of the CELM network consisting of three convolutional layers followed by a fully-connected layer.

Thus, a major difference between the ELM and the CELM models is the presence of the feature extraction step using random filters. The CELM model involves many more hyperparameters that can largely affect both the prediction error and the total number of trainable parameters in the network; in fact, the performance of the ELMs relies on the configuration of the fully-connected layer, whereas the performance of the CELMs is determined not only by the fully-connected layer but also by the architecture of the preceding

convolutional layers.

Regarding the optimization of the CELM model shown in Figure 4.2, we consider the integer parameters described in Table 4.2.

Table 4.2: Parameters of the CELM to be optimized and their bounds.

Parameter	Description	Min	Max
ch_1	number of filters in the first convolution layer	1	20
k_1	length of each filter of ch_1	1	20
ch_2	number of filters in the second convolution layer	1	20
k_2	length of each filter of ch_2	1	20
ch_3	number of filters in the third convolution layer	1	20
k_3	length of each filter of ch_3	1	20
L	number of hidden neurons in the fully-connected layer	1	80

Considering that the architecture parameters in the CELM are all integers, the genotype consists in this case of seven integer values. The first six are reserved for constructing the three convolutional layers, while the number of hidden neurons in the following fully-connected layer is determined by the remaining integer value L. Because CELMs with stacked convolutional layers (in particular three) have been shown to perform well in previous works[94,95], we fix the number of the convolutional layers to three. Instead, the parameters regarding the filters in each convolutional layer are encoded, such that each individual generated during the evolutionary search extracts different convolutional features. It should be noted that the parameters regarding the number of filters are multiplied by a fixed value of 10 when we generate the phenotype, while the parameters regarding the filter lengths are used as they are. The bounds (chosen empirically) for the seven parameters of the CELMs are shown in Table 4.2. Since we set the maximum value of L to $4,000$ for the ELMs in our previous work[14] (also summarized in Section 4.1.1), the same upper bound is considered also for the CELMs. As specified in Table 4.2, the

bounds of the parameter regarding the number of hidden neurons L are set to $[1, 80]$, however, this integer is multiplied by a fixed value of 50 when each genotype is translated into to corresponding phenotype. As explained in the case of the ELM optimization, the multiplicand is used to decrease the possible number of combinations determined by the bounds of the parameters: thanks to this discretization, we reduce the number of possible combinations in the search space from 2.56×10^{14} to 5×10^{9}.

Concerning the activation function, we use the sigmoid function for all the nodes in the fully-connected layer of the CELMs. This choice follows the existing works on CELMs[94,96,97], which all make use of the sigmoid function in the fully-connected layer, and the recent study[97] where the authors tested different widely-used activation functions, including the sigmoid and the hyperbolic tangent, and verified that the sigmoid can achieve the best prediction accuracy.

4.1.3 Convolutional neural network

CNNs have provided excellent performances on data-driven RUL prediction tasks[98,99] as well as time series processing tasks[100]. To distinguish from the CELM described above, it should be noted that the CNNs discussed in this subsection are BPNNs which are NNs trained by BP algorithms. In particular, a 1-D CNN has shown promising results comparable to RNNs[98,101] on the prognostics benchmark problem, despite its relatively simple backbone structure which includes 1-D convolutional layers and no pooling layers. Driven by the good results, we adopt a 1-D CNN as our backbone network, whose architecture should be optimized.

In detail, this network consists of stacked convolutional layers followed by a fully connected layer. Each convolutional layer involves a *filter* (i.e., a convolution matrix) to get the *filter response* of the *local receptive field*, a so-called *feature map*. Namely, the *local receptive field* refers to the part of

the input on which the filter slides across the horizontal axis indicating the temporal direction. A set of convolutional layers is responsible for extracting high-level feature representations, and the following fully-connected layer produces the output of the network (which corresponds to a RUL value in this work) from the extracted features.

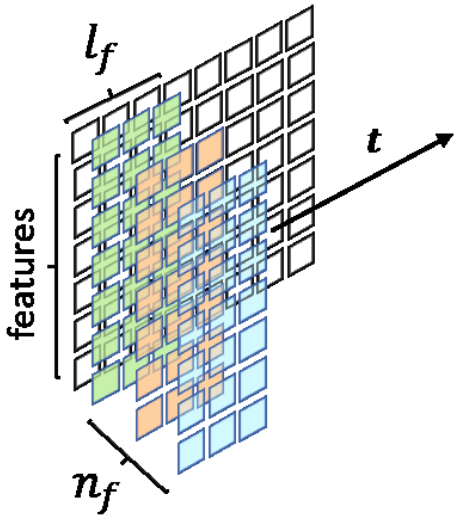


Figure 4.3: Illustration of 1-D convolution layer with n_f filters of length l_f.

Table 4.3: Architecture parameters of the 1-D CNN and their bounds.

Parameter	Description	Min	Max
n_l	number of convolution layers	3	8
n_f	number of filters in each convolution layer	5	25
l_f	length of convolution filters	5	25
$n_{f.c.}$	number of neurons in the fully connected layer	5	15

The architectural design of the 1-D CNN described above needs to choose the values of several hyper-parameters that mainly affect the prediction error and determine the network's total number of trainable parameters. More specifically, the number of convolutional layers is denoted as n_l, and a larger number of n_l offers a higher level of feature representation by the stacked convolutional layers, but this increases the number of parameters of the CNN.

Each convolutional layer has two hyper-parameters regarding the convolution filter, n_f and l_f; each of n_f filters of length l_f slides over its input features to apply convolution in the temporal direction as shown in Figure 4.3. The number of neurons in the fully connected layer, $n_{f.c.}$, works on the regression task based on the extracted feature. Thus, we consider the optimization of the four architecture parameters summarized in Table 4.3.

The search space to be explored by our optimization algorithm is defined by the parameters and their bounds specified in the table. The lower and upper bounds for each parameter have been determined empirically; we do not include the smaller networks containing too few trainable parameters in our search space since their learning capability is not enough to decrease the training loss; the larger networks comprising too many parameters are not considered as well because they may overfit the training data (i.e., they can decrease the training loss but not the validation loss). Note that the number of neurons in the fully connected layer is divided by 10 when we encode the solutions so that the number of neurons can be up to 150, but there is a much smaller number of integers within the bounds for $n_{f.c.}$. On top of those considerations, we chose specific values such that approximately $30,000$ 1-D CNN configurations exist in the search space.

4.2 MOO algorithm

When it comes to the optimization of the networks described in Section 4.1, we consider a MOO approach to identify their optimal architectures achieving the best trade-off between RUL prediction error and the number of trainable parameters. In our work, NSGA-II is used to get a Pareto-optimal front.

Each run of the evolutionary search starts with randomly initializing a population of n_{pop} individuals. Every individual in the population should contain each objective value that can be obtained by means of the evaluation

specified in Section 3.2; we do not employ performance prediction methods for the ELMs and CELMs since their training is extremely fast in essence, while the two methods introduced in Sections 3.3.2 and 3.3.3 are used when we evaluate the RUL prediction error (the validation RMSE) of the CNNs.

In the main loop of the algorithm, an offspring population of the same size is created by using tournament selection, crossover, and mutation. The tournament selection primarily sorts a population according to the level of non-domination. Once all the non-dominated individuals have been considered, the density estimation of solutions by the so-called crowding distance[17] is then taken as the secondary criteria which promote individuals that lie in less crowded areas of the Pareto front.

Regarding genetic operators, we use 1-point crossover and uniform mutation. The probability of crossover and mutation is set to 0.5 (i.e., $p_{cx} = p_{mut} = 0.5$) for the same value and reason as stated in Section 3.4. The expected number of mutations per individual is determined by the probability p_{gene} that indicates the probability of applying the mutation operator to a single gene. p_{gene} is set to 0.4, which can induce, on average, a reasonable number of mutated genes so that we can suppress too small or too destructive mutations; on average of 1.2 mutated genes are expected out of 3 for the ELM discussed in Section 4.1.1; 2.8 mutated genes are expected out of 7 for the CELM discussed in Section 4.1.2; 1.6 mutated genes are expected out of 4 for the CNN shown in Section 4.1.3. In turn, this allows us to have a relatively faster architecture search process and avoid disruptive mutations.

Returning to the elucidation of the main loop, the new individuals are pooled with the parents. The combined population is then sorted according to non-domination. The best non-dominated sets are inserted into the new population until it is as close as possible to the preset population size n_{pop} but does not exceed it. 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. This loop is terminated after a fixed number of generations n_{gen}; the evolutionary algorithm returns a Pareto front, which is defined as the set of trade-off solutions at the top dominance level.

Chapter 5

Experiments

In this research work, we hypothesize that applying NAS can result in better neural architectures for the data-driven RUL prediction task. We validate this a set of numerical experiments described in this chapter. The experiments are carried out on the two benchmarks. we use the CMAPSS to test our evolutionary NAS on the DL models presented in Chapter 3,i.e., the architecture optimization of the multi-head CNN-LSTM and Transformers is tested on the CMAPSS. While, we use the N-CMAPSS to test the MOO of the ELM and CNN architecture introduced in Chapter 4. Finally, the MOO of the CELM architecture is tested on the CMAPSS. The used datasets are described in Section 5.1. In Sections 5.2 and 5.3, we detail the evaluation metrics considered in our work and the computational setup along with the training details of our experimentation, respectively. Finally, Section 5.4 presents the experimental results and their comparative analysis; the performance of the optimized networks is compared with the currently used RUL prediction models.

5.1 Benchmark dataset

In this section, we first introduce the CMAPSS dataset which has been used in our previous works introducing the single-objective optimization (SOO) of

DL-based RUL prediction models[12,13]. This is followed by the description of the N-CMAPSS dataset, in which we develop the multi-objective RUL prediction models as presented in papers[14,16]. The work based on paper[15], which is an extension of a previous paper[14], uses the CMAPSS dataset for its evaluation.

5.1.1 CMAPSS

Accurate RUL predictions for complex systems make it possible to develop a smart maintenance policy, which in turn, reduces any unplanned downtime and cuts dispensable losses. The airline industry is a typical instance of this problem since timely maintenance of aircraft engines can largely affect the overall operation cost. By predicting the engines' RUL accurately, we can minimize maintenance costs.

Considering the above, NASA has introduced the CMAPSS dataset, comprising some run-to-failure trajectories simulated with the CMAPSS simulator[102]. It simulates the degradation of a commercial turbofan engine, depicted in Figure 5.1 and provides the recorded sensor measurements under different settings of health-related parameters[1].

According to the seminal paper[18] introducing the dataset, the data have been collected by the CMAPSS operation in closed-loop configurations. The CMAPSS produces various sensor response surfaces as its outputs from a set of health-parameter inputs that enable users to simulate the effects of faults in the following give rotating components which are crucial components associated with the engine's failure: fan, low-pressure compressor (LPC), high-pressure compressor (HPC), high-pressure turbine (HPT), and low-pressure turbine (LPT). The outputs produced by the CMAPSS simulation corresponding to the inputs are used as the data for our experiments.

The dataset consists of four sub-datasets: FD001, FD002, FD003, and FD004. Each of them considers different operating states and fault modes

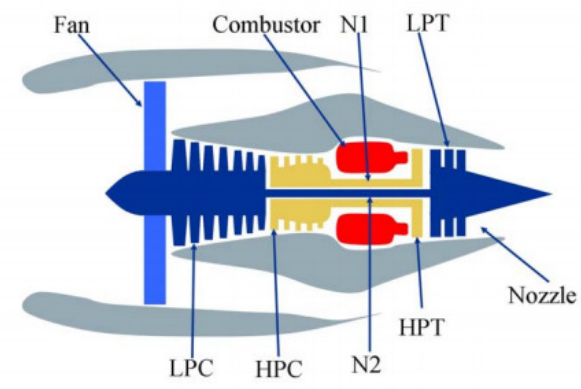


Figure 5.1: Simplified diagram of the turbofan engine simulated in CMAPSS [1].

when it generates 21 output variables that represent the degradation trajectories of recordings from different sensors. It should be noted that the update for the RUL prediction and the sensor measurement occurs at every *cycle*, which is the time unit considered in this dataset.

As summarized in Table 5.1, each sub-dataset comprises of a training set D_{train} and a test set D_{test}; the former provides a running history of each engine until its failure, while a history of each test engine on D_{test} ends at a certain cycle before failure. Thus, the CMAPSS dataset enables the task of accurately predicting the RUL of each test engine at the end of its given history, for which it is allowed to exploit the data in D_{train}.

It is more challenging to make accurate predictions on FD002 and FD004 than on FD001 and FD003 because the former two are simulated under six different operating conditions, while the latter two are simulated under only one condition. The RUL prediction error on FD003 (FD004) tends to be generally higher than the error on FD001 (FD002) in general. This heppens because it is more difficult to develop a pure data-driven model when a dataset contains multiple failure modes compared to a single failure mode.

In this dataset, we discard the 7 time series whose data points never vary over time out of the 21 trajectories and take into account the remaining 14 time series.

Table 5.1: CMAPSS dataset overview.

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

5.1.2 N-CMAPSS

In 2021, NASA's data repository released the N-CMAPSS dataset, which contains data acquired under real flight conditions. This large realistic dataset is used to test the MOO of different neural architectures suggested in Section 4.1.

The previous dataset, the CMAPSS dataset explained in Section 5.1.1, has been widely used to develop and evaluate the RUL prediction models after it became publicly available on NASA's data repository in 2008. It is solely based on MATLAB simulations, without considering real flight conditions, so that each time series is rather short (just a few hundred samples). On the other hand, in the new realistic dataset, each time series consists of millions of samples by reflecting real flight conditions, thus the total size of the dataset is significantly larger. Therefore, the N-CMAPSS dataset provides a chance to develop reliable algorithms for RUL prediction in a real-world context. Moreover, while the previous dataset was small enough to allow researchers to focus only on the minimization of the RUL prediction error, without considering the training time, the new dataset, due to its much larger amount of data, requires algorithms that are faster to train, without compromising the RUL prediction error.

The experimentations of the MOO algorithm specified in Section 4.2 for

the ELM (Section 4.1.1) and 1-D CNN (Section 4.1.3) are based on the N-CMAPSS dataset. Specifically, we only use its sub-dataset DS02, which has been developed for data-driven methods[19]. It consists of the run-to-failure degradation trajectories of nine turbofan engines with unknown and different initial conditions. The synthetic trajectories were generated with the CMAPSS dynamic model implemented in MATLAB, but a fidelity gap between simulation and reality is mitigated by reflecting real flight conditions recorded on board a commercial jet. Furthermore, the relation between the degradation and its operation history is considered, to extend the degradation modeling[19]. 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}. In particular, the u_{14} and u_{15} relate to shorter and lower altitude flights compared to those of the training units, so that the evaluation results on the D_{test} can implicitly reflect the generalization capability of the RUL prediction model.

Table 5.2 describes each unit in the dataset w.r.t the number of samples m_i, the end-of-life time t_{EOL} and the failure modes. The total number of samples (i.e., timestamps) is 5.26M in D_{train} and 1.25M in D_{test}, with a sampling rate of 1Hz. The end-of-life time t_{EOL} points out the counted flight cycles at the end of the engine's lifespan. There are two distinctive failure modes in the dataset: abnormal HPT and LPT. The combination of the two failure modes for a unit means that the unit is subject to a more complex failure mode than a single-failure mode.

The dataset provides condition monitoring signals that are related to the useful life of the flight engine. Following the setup used in[98], we select the same 20 signals. The multivariate time series from the 20 signals is used as an input for the networks, therefore the dimension of the input sample d is 20.

Table 5.2: Overview of each unit in the DS02 of N-CMAPSS dataset.

Training set (D_{train})				Test set (D_{test})			
Unit	m_i(M)	t_{EOL}	Failure Mode	Unit	m_i(M)	t_{EOL}	Failure Mode
u_2	0.85	75	HPT	u_{11}	0.66	59	HPT+LPT
u_5	1.03	89	HPT	u_{14}	0.16	76	HPT+LPT
u_{10}	0.95	82	HPT	u_{15}	0.43	67	HPT+LPT
u_{16}	0.77	63	HPT+LPT				
u_{18}	0.89	71	HPT+LPT				
u_{20}	0.77	66	HPT+LPT				

5.2 Evaluation metrics

In the SOO tasks[12,13], the objective is to reduce the RUL prediction error, where it is defined as the discrepancy between the predicted and target RUL. Particularly, the prediction error is quantified w.r.t. two metrics: the RMSE and the s-score[18].

Let the error between the predicted and target RUL be $d_i = RUL_i^{predicted} - RUL_i^{target}$, where d_i denotes the error on the i-th sample. The RMSE on D_{test} is then defined as follows:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n} d_i^2} \tag{5.1}$$

where n represents the number of samples in D_{test}. The RMSE is considered as one of the metrics considering that it has the benefit of weighting large errors more; penalizing large errors should be considered since the errors in the context of the RUL prediction are related to cost and safety. Although mean absolute error (MAE) has not been considered as a metric in the following experiments, it can be very useful to calculate the simple MAE when the proposed strategies are applied to industrial applications, so that obtained results can be interpreted easily and intuitively.

Furthermore, we consider an additional metric, called s-score, which penalizes optimistic RUL predictions. More specifically, it separates the predic-

tions according to whether they are "optimistic" or "pessimistic" via an asymmetric function which is formulated to grant smaller values to pessimistic predictions compared to optimistic predictions. This is formulated in this way:

$$s\text{-score} = \sum_{i=1}^{n} s_i, \quad s_i = \begin{cases} e^{-\frac{d_i}{13}} - 1, & d_i < 0 \\ e^{\frac{d_i}{10}} - 1, & d_i \geq 0 \end{cases} . \tag{5.2}$$

Hence, different from the RMSE, the s-score evaluates the risk that the output of the network is larger than the actual RUL value. It should be noted that we use the s-score solely for evaluating the methods on the test set; on the other hand, we perform the evolutionary optimization on the RMSE, since it provides more information from an optimization point of view w.r.t. the s-score. In fact, based on our work[12], networks optimized using the RMSE as fitness function provide better results in terms of both metrics, compared to networks optimized based on the s-score.

Regarding the MOO tasks, our goal is to find solutions to achieve the trade-off between the number of trainable parameters and prediction error. Specifically, in the experiments on the ELM and CELM optimization[14,15], we evaluate the performance of the solutions in terms of the two objective values in the test set by comparing them with the results obtained by the different methods under study. For each solution, the test RMSE (Equation (5.1)) and the number of trainable parameters are calculated. We then consider the averages of each of the two calculated values of the solutions as the final results used for the comparative analysis.

In the paper that introduce a novel way to accelerate the NAS[16], our experimentation aims to assess how the speed-up techniques affect the GA in terms of two metrics: 1) the quality of the solutions, represented by the hypervolume (HV) where the HV is calculated on the space defined by the test RMSE and the number of trainable parameters, and 2) the GA runtime, in GPU hours.

5.3 Computational setup and training details

The multi-head CNN-LSTM (Section 3.1.1) and 1-D CNN networks (Section 4.1.3) are implemented using TensorFlow 2.4, while we use the high-performance toolbox[1] for ELM to implement the ELM (Section 4.1.1). The CELMs (Section 4.1.2) and Transformer networks (Section 3.1.2) are implemented in PyTorch. The baseline GAs (Sections 3.4 and 4.2) are implemented using the evolutionary computation framework DEAP[103]. Our codes are available online[2]. All the experiments have been carried out on a single NVIDIA Titan Xp GPU.

Both datasets used in our work consist of multivariate time series, and the value of each time series is rescaled into the range $[-1, 1]$ by using min-max normalization so that we prepare the input samples for the networks. The data preparation and training details for the different BPNN models are specified in the following subsections.

5.3.1 Training details for the multi-head CNN-LSTM

The information summarized in this subsection is reported in detail in our paper[12]. To create the input samples for the DL network, we apply time window having fixed length T (i.e., it determines the sequence length l_s explained in Section 3.1.1). Following the window size suggested in[56], the value of T is set to 31, 21, 38, and 19 respectively for FD001, FD002, FD003, and FD004.

By leveraging the prepared input sequences labeled with RUL, we conduct supervised learning. The loss function of the network is the MSE as defined by Equation (3.1). The weights of the model are then optimized to minimize the loss using the RMSprop algorithm. Based on the empirical evidence from

[1] `https://github.com/akusok/hpelm`
[2] `https://github.com/mohyunho`

our previous work[68], we set the batch size of gradient descent to 400, and we limit the maximum number of training epochs to 20.

As described in Section 3.3.1, we introduce an early-stopping mechanism in the optimizer along with the learning-rate decay. In particular, the tolerable amount of epochs with no improvement in the validation loss, so-called patience, is 5. The learning-rate starts with a value of 10^{-4} and drops to 10^{-5} after 10 epochs, then we divide it again by 10 after 5 epochs.

An additional aspect to highlight is that, in general, reproducible results might be needed in some industrial contexts. Nevertheless, some operations of the DNNs implemented by the 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 floating point representation. To get reproducible results, we considered using the determinism library[3] that provides deterministic outputs by addressing the issues above. However, we noted that the determinism library dramatically slows down the GPU computation. Considering the too long GA runtime with determinism, we decided not to use it. Instead, we repeat the ENAS-PdM process without determinism three times for each sub-dataset; each evolutionary run gave the best architecture yielded from the search, and the performance average of the collected solutions was taken as the final result.

5.3.2 Training details for the Transformers

The details discussed in this section cover our journal paper[13]. We prepare the input sequence for the Transformer by applying a fixed-length time window with stride 1 to the normalized time series data. The size of the windows for FD001, FD002, FD003 and FD004 is set to 40, 60, 40 and 60, respectively, following the values suggested in[74].

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

The loss function we employed is the MSE (see Equation (3.1)). The training of the network then corresponds to finding the network weights which minimize the loss by using the Adam optimizer[104]. The batch size is set to 256, and this size is also used for calculating the SNIP value when we prepare the predictor inputs. Similar to Section 5.3.1, we apply early-stopping to the training of the Transformer as well; network training stops early if the optimizer fails to improve the validation loss within a given patience; otherwise, the training continues until the pre-defined maximum number of epochs is reached. The value of patience is 10. We train each network for at most 100 epochs on the FD001 and FD003, and 200 epochs on the FD002 and FD004. The reason for setting a larger value for the latter two sub-datasets is that they have a greater number of training samples than the former two. Thus, on FD002 and FD004, relatively more training epochs are required for allowing the learning curve to reach a plateau.

We apply a learning-rate decay in conjunction with the early-stopping mechanism; the learning-rate starts with a value of 10^{-5}, and it is then multiplied by the constant 0.9 for every 10 epoch. This contributes to suppressing the fluctuation of the validation loss curve, so that we can avoid misleading observations given by those fluctuations.

5.3.3 Training details for the 1-D CNN

As shown in Figure 4.3, the 1D-CNN requires time-windowed data as an input to apply 1-D convolution in the temporal direction; we apply a time window of length 50 and stride 50 so that the given multivariate time series consisting of the 20 signals in the N-CMAPSS dataset is divided into input samples, with each sample of size 50×20.

For training, we use stochastic gradient descent (SGD). In particular, AMSgrad[104] is used as an 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[12]. 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 zero-cost proxy, also called architecture score, introduced in Section 3.3.3. We randomly choose 512 samples from D_v, and use it as the mini-batch X. In this regard, the ablation study in[80] verified that the choice of the mini-batch has little impact on the architecture score trend over different network architectures.

5.4 Experimental results

5.4.1 Evolutionary NAS on DL models

Multi-head CNN-LSTM

One of the evolutionary NAS approaches introduced in Section 3.4, the ENAS-PdM, aims to optimize the architecture of the multi-head CNN-LSTM that has been manually designed in our previous work[68], so that we can improve the RUL prediction accuracy when we deploy the multi-head CNN-LSTM as a RUL prediction tool.

We organize the experiments in the following way to verify if the model found by ENAS-PdM on a certain sub-dataset also provides promising results on the others; we choose one sub-dataset and run the evolutionary search using the sub-dataset to obtain an architecture optimized on it, then the found architecture is tested on the other three sub-datasets as well as the sub-dataset used for the search.

The fitness function used in the experiments discussed in this section is validation RMSE. In the full experiments reported in the paper[12], we have used not only the validation RMSE but the s-score as our fitness function to verify if choosing one of the two different metrics as the fitness function affects the test results; the experimental results have shown that the optimized

networks based on the validation RMSE yield better test results. Therefore, in the rest of this section, we present the experimental results taking the RMSE as the fitness function.

Due to the non-deterministic GPU operations discussed in Section 5.3.1, we run ENAS-PdM three times on each sub-dataset under exactly the same settings to show the reliability of the optimization process. Eventually, we collect the best architectures discovered by ENAS-PdM on the four sub-datasets at the end of the 12 total runs. As shown in Table 5.3, the performance of the network having the best architecture given by ENAS-PdM is evaluated on all the sub-datasets in terms of the test RMSE and s-score, regardless of which sub-dataset was used for the optimization conducted by our EA.

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 (i.e., $n_{pop} = n_{gen} = 50$), which allows enough evaluations to ensure the convergence of 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. We also count the number of evaluations across the EA runs to measure the saved computational costs of ENAS-PdM. As can be seen in Table 5.5, 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 based on the so-called history mechanism explained in Section 3.4. 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 find good solutions while evaluating less than 4.3% of the 20400 possible combinations described in Table 3.1.

Table 5.3 provides the test results of the best architecture found by every

Table 5.3: Results of the best architectures found by ENAS-PdM in terms of test RMSE and *s*-score performance on the CMAPSS dataset. The mean and standard deviation (SD) of the performance values in each column is selected for comparison to the state-of-the-art methods in Tables 5.7 and 5.8.

Used sub-dataset for EA	EA runs	RMSE				*s*-score			
		FD001	FD002	FD003	FD004	FD001	FD002	FD003	FD004
FD001	1st run	11.48	17.47	12.48	20.59	240	2074	412	3592
	2nd run	11.54	17.79	12.10	20.93	250	2260	234	3908
	3rd run	11.71	17.55	12.88	20.38	254	1695	564	3395
FD002	1st run	12.01	17.76	13.24	19.73	263	5158	926	3004
	2nd run	12.55	18.20	12.81	20.46	292	5740	425	4027
	3rd run	12.57	18.48	11.68	21.40	302	7124	241	5163
FD003	1st run	12.16	17.83	12.82	20.53	280	1766	498	4062
	2nd run	13.06	18.85	14.11	20.78	295	6610	1053	4085
	3rd run	11.76	17.84	12.64	20.58	271	2641	408	3613
FD004	1st run	11.39	17.69	13.74	20.02	224	5170	1028	2897
	2nd run	12.49	17.70	13.76	18.97	246	4052	834	2712
	3rd run	11.39	17.69	13.74	20.02	224	5170	1028	2897

Table 5.4: Specifications of the multi-head CNN-LSTM architectures discovered by ENAS-PdM and their performance in terms of the sum of test RMSE and *s*-score.

Used sub-dataset for EA	EA runs	Phenotype (l_w, m, l_f, C, $L_1 \cdot 20$, $L_2 \cdot 20$)	RMSE (sum)	*s*-score (sum)
FD001	1st run	(2, 3, 2, 1, 400, 260)	62.02	6318
	2nd run	(2, 3, 2, 1, 400, 240)	62.36	6652
	3rd run	(2, 5, 2, 1, 300, 240)	62.52	5908
FD002	1st run	(1, 9, 1, 1, 160, 140)	62.74	9351
	2nd run	(1, 8, 1, 1, 260, 260)	64.02	10484
	3rd run	(1, 7, 1, 1, 320, 260)	64.13	12830
FD003	1st run	(2, 4, 2, 1, 260, 240)	63.34	6606
	2nd run	(2, 3, 2, 2, 280, 240)	66.80	12043
	3rd run	(2, 3, 2, 1, 360, 300)	62.82	6933
FD004	1st run	(2, 10, 2, 2, 100, 100)	62.84	9319
	2nd run	(1, 9, 1, 2, 80, 80)	62.92	7844
	3rd run	(2, 10, 2, 2, 100, 100)	62.84	9319

independent run of ENAS-PdM using the validation RMSE as the fitness function. The specification of the multi-head CNN-LSTM architectures (i.e., Phenotype) is clarified in Table 5.4. For the three independent runs on each

Table 5.5: Average number of evaluations across 3 independent ENAS-PdM runs.

Sub-datasets	FD001	FD002	FD003	FD004
Number of evaluations (avg. across 3 independent GA runs)	869/2500	830/2500	837/2500	656/2500

sub-dataset, we indeed get three different architectures. Although the architectures are different, their performance is similar. Moreover, an architecture optimized for one sub-dataset also provides promising results on the others. This reveals that the solutions based on a certain sub-dataset can be generalized to other sub-datasets.

Thus, the proposed method predicting RUL involves all 12 networks having the best architectures found by the 12 independent GA runs. The performance of the proposed method is then calculated as mean $\pm$ standard deviation (SD) across 12 independent runs of the evolutionary search, regardless of the sub-dataset used for the EA. Namely, for each sub-dataset, the mean and SD of the 12 performance values shown in Table 5.3 are eventually considered as the results of our proposed method and compared against the state-of-the-art.

Tables 5.7 and 5.8 display the compared methods and their results in terms of the test RMSE and s-score respectively. The methods in the tables are reported in chronological order of publication. In the tables, the second last row shows the final results of the proposed method; the mean and SD of the values in each column of Table 5.3

The first method (CNN)[44] is a conventional Feed-Forward NN with two convolutional layers. The following two rows prove that standard LSTM[48] and Bi-directional LSTM (BiLSTM)[48] outperform the CNN. The next method, DCNN[99], provides lower RMSE w.r.t. the methods using LSTM. The next six methods have been proposed more recently. In[105], a RNN-based deep architecture is used for RUL prediction under a semi-supervised

Table 5.6: Specification of the best architectures found in each of the 5 EA runs, in conjunction with their test RMSE and s-score performance. For each sub-dataset, the mean and SD of the performance reported in the table are selected for comparison to the state-of-the-art methods in Tables 5.7 and 5.8.

Used sub-dataset for EA & test	EA runs	Phenotype (Transformer architecture)											Performance	
		d_{model} [24, 100]	d_k [24, 100]	d_v [24, 100]	d_{ff_s} [24, 100]	d_{ff_t} [24, 100]	d_{ff_d} [24, 100]	h_s [1, 16]	h_t [1, 16]	h_d [1, 16]	N_{enc} [1, 3]	N_{dec} [1, 3]	RMSE	s-score
FD001	1st run	92	24	84	84	24	80	16	4	14	3	3	11.50	202
	2nd run	100	92	88	88	52	80	15	2	6	3	3	11.89	230
	3rd run	100	92	88	24	36	72	10	15	16	3	3	11.41	193
	4th run	100	48	72	40	32	60	14	10	16	3	3	11.76	236
	5th run	88	28	92	40	44	68	14	5	9	3	3	11.60	217
FD002	1st run	96	64	52	96	92	68	6	16	11	2	2	16.14	1131
	2nd run	100	84	28	76	100	88	3	11	8	3	3	15.42	997
	3rd run	92	100	28	24	80	44	8	11	12	2	3	16.26	1233
	4th run	92	96	28	32	92	24	1	16	3	3	1	16.35	1163
	5th run	100	92	56	68	48	72	12	4	13	2	3	15.80	1145
FD003	1st run	92	84	92	32	45	40	8	2	15	3	1	11.35	227
	2nd run	100	100	52	64	32	68	16	13	16	2	1	11.31	226
	3rd run	88	84	68	80	44	56	14	2	15	2	2	11.15	230
	4th run	96	96	92	52	48	60	16	4	11	3	3	11.39	218
	5th run	88	52	92	60	88	72	12	12	12	3	3	11.57	241
FD004	1st run	84	76	92	92	96	40	2	10	15	3	1	20.00	2298
	2nd run	96	68	60	100	76	96	2	8	16	3	1	19.85	3038
	3rd run	100	76	100	40	32	60	1	16	14	3	3	20.18	2602
	4th run	100	80	24	56	84	56	3	16	5	3	2	20.70	3109
	5th run	92	100	28	76	68	40	5	11	16	3	3	20.03	2315

setup, and a GA approach is used to tune its training hyper-parameters, rather than its architecture. The directed acyclic graph (DAG) network[56] 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[68], which uses a handcrafted multi-head CNN-LSTM. Then, the DCNN with adaptive batch normalization (AdaBN)[106] achieves the lowest values in both metrics compared to the previous methods. The method based on an RNN autoencoder scheme[58] constructs health index curves showing degradation to predict RUL, using a bidirectional RNN-based autoencoder

Table 5.7: Comparison of RUL prediction performance of the networks found by the evolutionary NAS with state-of-the-art methods (sorted by year), in terms of test RMSE.

Method	RMSE				
	FD001	FD002	FD003	FD004	Sum
CNN, 2016 [44]	18.45	30.29	19.82	29.16	97.72
LSTM, 2017 [48]	16.14	24.49	16.18	28.17	84.98
BiLSTM, 2018 [48]	13.65	23.18	13.74	24.86	75.43
DCNN, 2018 [99]	12.61	22.36	12.64	23.31	70.92
Semi-supervised DL, 2019 [105]	12.56	22.73	12.10	22.66	70.05
DAG network, 2019 [56]	11.96	20.34	12.46	22.43	67.09
Multi-head CNN-LSTM, 2020 [68]	13.27	19.49	13.21	23.89	69.86
AdaBN-DCNN, 2020 [106]	11.94	19.29	12.31	22.14	65.68
RNN+AE, 2020 [58]	13.58	19.59	19.16	22.15	74.48
AGCNN, 2020 [59]	12.42	19.43	13.39	21.50	66.74
ENAS-PdM on Multi-head CNN-LSTM, 2021 [12]	11.96±0.59	17.94±0.40	13.01±0.73	20.36±0.62	63.27
CNN+attention, 2021 [107]	**11.48**	17.25	12.31	20.58	61.62
Surrogate-assisted EA on Transformer, 2023 [13]	11.63±0.19	**15.99±0.38**	**11.35±0.15**	**20.15±0.32**	**59.12**

Table 5.8: Comparison of RUL prediction performance of the networks found by the evolutionary NAS with state-of-the-art methods (sorted by year), in terms of s-score.

Methods	s-score				
	FD001	FD002	FD003	FD004	Sum
CNN, 2016 [44]	1290	13600	1600	7890	24380
LSTM, 2017 [48]	338	4450	852	5550	11190
BiLSTM, 2018 [48]	295	4130	317	5430	10172
DCNN, 2018 [99]	274	10400	284	12500	23458
Semi-supervised DL, 2019 [105]	231	3370	251	2840	6692
DAG network, 2019 [56]	229	2730	553	3370	6882
Multi-head CNN-LSTM, 2020 [68]	330	2880	401	6520	10131
AdaBN-DCNN, 2020 [106]	220	2250	260	3630	6360
RNN+AE, 2020 [58]	228	2650	1727	2901	7506
AGCNN, 2020 [59]	225	1492	**227**	3392	5336
ENAS-PdM on Multi-head CNN-LSTM, 2021 [12]	262±27	4120±1143	637±280	3612±696	8613
CNN+attention, 2021 [107]	**198**	1144	251	**2072**	**3665**
Surrogate-assisted EA on Transformer, 2023 [13]	215±18	**1133±85**	228±8	2672±386	4248

scheme as a feature extractor. The AGCNN [59] is a custom encoder-decoder architecture in which the encoder is made up of a bidirectional RNN and a CNN.

The row directly below them presents 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[68]. Regarding the s-score performance, the proposed method does not offer outstanding results compared with the recent methods, but we can verify that the architecture optimization with ENAS-PdM improves RUL prediction performance in terms of the sum of s-score, compared to manually designing the network[68]. Most importantly, the proposed method outperforms any other DL-based method developed manually by human experts in terms of the test RMSE, except for the methods employing attention mechanisms that have been proposed after our proposal. Hence, our results considerably advance the state-of-the-art RUL predictions in terms of the test RMSE.

Transformers

Our experiments aim to find the optimal Transformer architectures from the search space specified in Section 3.1.2 using the surrogate-assisted evolutionary search proposed in Section 3.4 and to numerically evaluate the quality of the discovered solutions by calculating the two metrics defined in Section 5.2 on the test set D_{test} after training. Furthermore, we perform a comparative analysis contrasting the results from the proposed GA with the state of the art. Note that the experimental results presented below are reported in our journal paper[13].

The experiments begin by initializing the model-based performance predictor described in Section 3.3.4. The budget used to initialize the predictor, m, is determined as 100, i.e., we sample 100 different architectures by LHS, and those networks are fully trained on D_w (see Section 3.2) to observe the validation RMSE on D_v. After the observations, we train the predictor to minimize the error of the regression model output. We set m to a small value compared with the search space size, considering the following reasons: 1) as our predictor, we employ NGBoost, a probabilistic regression that per-

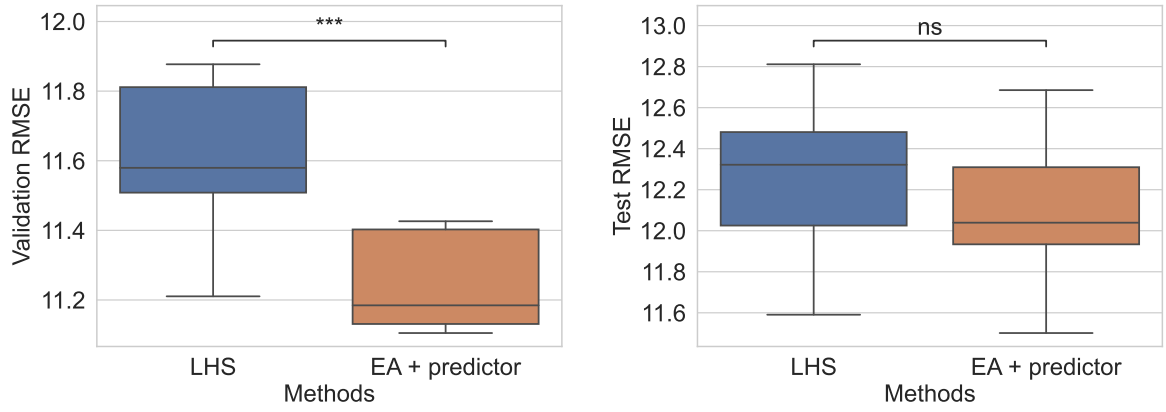
forms well with relatively small datasets[82]; and 2) for each generation, the predictor is updated with few samples which are expected to have better fitness.

After the initialization, we execute the proposed algorithm for a maximum number of generations n_{gen} of 10 with a population size n_{pop} of 1000. In the fitness evaluation step for each generation, we retrain at most only 1% of the population, i.e., the number of elites, k, is set to 10. Therefore, in each evolutionary run, the maximum possible number of network training processes is 100 over 10 generations, but we observed that the actual number of training processes conducted in our experiments were 61, 81, 77 and 73, respectively for each sub-dataset, thanks to the history that enables to reuse the fitness values observed in the previous generations. Thus, for each evolutionary run, we merely need to train at most 200 networks (100 for the initialization and 100 for the updates) out of approximately 2.36×10^{12} possible networks in the search space. When it comes to the definition of the observation history, as discussed before, this is an archive of all the solutions found during the search.

Before comparing our results to the state of the art, we compare the quality of the solutions obtained from a single run of the algorithm with LHS solutions to demonstrate the advantage of our algorithm; we define the solutions obtained by the proposed algorithm as the 10 best networks (in terms of fitness) from the history in a single run of evolutionary search. Those are then compared against the LHS solutions that are the 10 best LHS samples out of 100 in terms of the validation RMSE, so that we make a fair comparison with our approach by considering the same number of solutions. These comparisons are depicted in the box plots shown in Figures 5.2 to 5.5, respectively for each of the four CMAPSS sub-datasets. The statistical difference between the two solution sets is assessed based on the Mann-Whitney U (MWU) test[108]. For each pairwise comparison, we compute the statistical

significance and add a statistical annotation w.r.t. the MWU test p-value obtained on the two corresponding box plots.

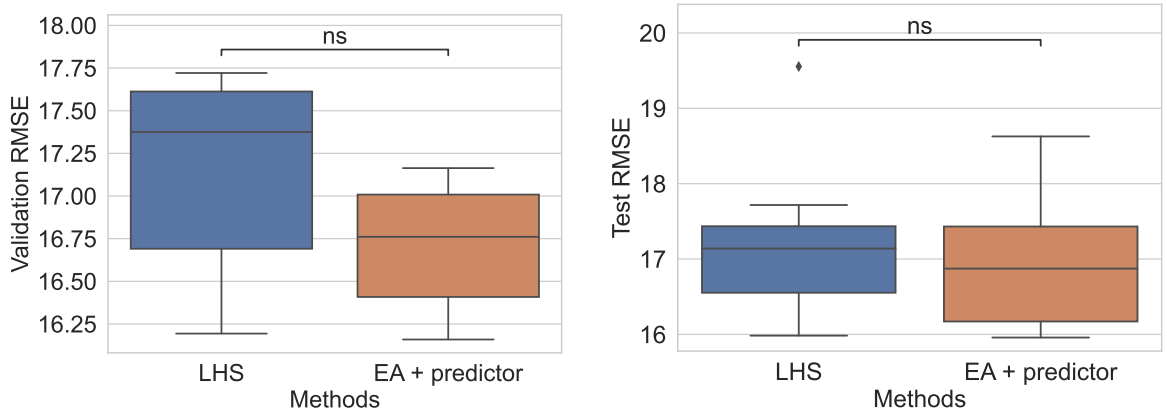
Regarding the validation RMSE, which is the objective of our optimization algorithm, the proposed method has statistically better performances compared to LHS, except for FD002 where the difference is not significant ("ns"). In particular, the p-values obtained on FD001 and FD003 are lower than 0.01, showing that the results we obtained are significantly better than LHS. On FD002, even though the difference is not significant, Figure 5.3 graphically demonstrates that the improvement is non-trivial; instead, the difference for FD004 is significant, but as Figure 5.5 reveals, it has a higher p-value than the one obtained for FD001 and FD003.



Validation RMSE **Test RMSE**

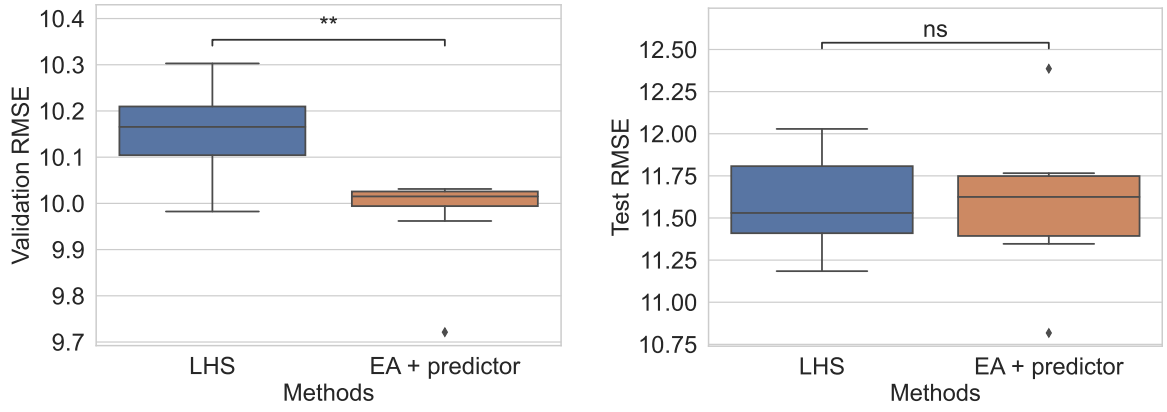
Figure 5.2: Box plots of the quality of solutions found by LHS and by the surrogate-assisted ENAS for Transformers on FD001.

The proposed GA and the LHS are also evaluated on the test set D_{test}. This performance comparison appears on the right side of Figures 5.2 to 5.5. We can observe that our method is better compared to LHS, in terms of test RMSE, i.e., the best solution obtained by the proposed algorithm can always provide at least one solution with lower test RMSE than any of the LHS solutions. Nevertheless, the performance differences between the 10



Validation RMSE **Test RMSE**

Figure 5.3: Box plots of the quality of solutions given by LHS and by the surrogate-assisted ENAS for Transformers, on FD002.

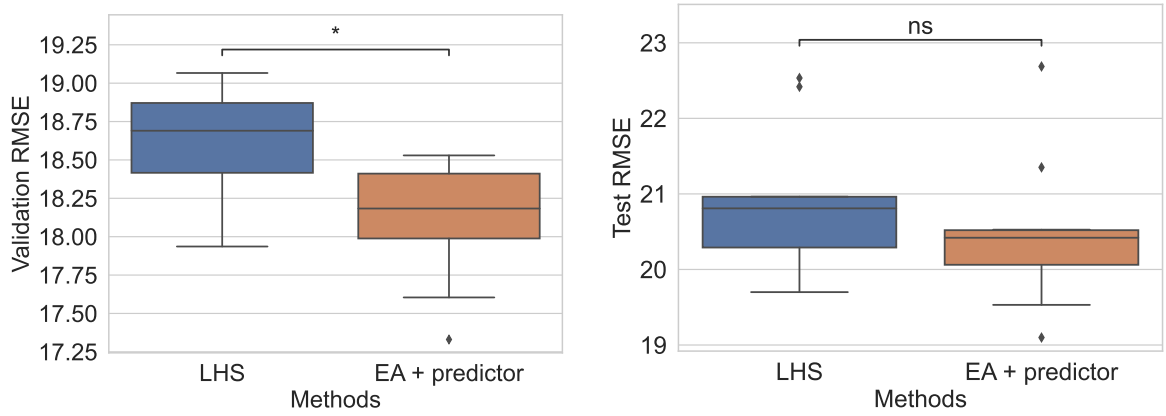


Validation RMSE **Test RMSE**

Figure 5.4: Box plots of the quality of solutions given by LHS and by the surrogate-assisted ENAS for Transformers, on FD003.

solutions found by the two methods are not statistically significant.

After the demonstration based on a single run of the EA, we perform 4 additional runs and take one best solution for each run (5 independent runs and 5 solutions in total). By considering multiple runs, we can enhance the reliability of the experimental results in terms of performance when our numerical results are compared to the results of other existing works. Specifically, for



Validation RMSE **Test RMSE**

Figure 5.5: Box plots of the quality of solutions given by LHS and by the surrogate-assisted ENAS for Transformers, on FD004.

each sub-dataset, we execute 5 independent runs of the GA initialized with different random seeds and take the best (in terms of fitness) architecture along with its performance for each run. Table 5.6 describes all the obtained solutions and their performance on the four sub-datasets respectively. For each sub-dataset, The performance of the proposed method is then calculated as the mean and SD of the 5 performance values in terms of the test RMSE and s-score. Those final results of the comparative analysis are reported in Tables 5.7 and 5.8; for each method, the results on each sub-dataset are independent. The last column of each table contains the sum of the results across the four sub-datasets, to obtain an aggregate measure of the robustness of the compared methods.

The first 11 methods in the tables, from the CNN[44] to our ENAS-PdM[12], are already described in the above, the first half of Section 5.4.1. The latest compared method[107] applies an attention mechanism on top of the features extracted by four convolutional layers.

Compared to the other works described above, developing our method requires less knowledge and effort from human experts, because our method

automatically discovers optimal architectures for RUL prediction. In the resulting tables, each last row reports the experimental results obtained by the best solutions found with our method. With regards to the sum of test RMSE values, the proposed method outperforms the compared methods, which have all been manually developed by human experts. Even compared to ENAS-PdM[12] which is proposed in our previous work that also uses evolutionary NAS, the proposed method[13] gives better results. Moreover, on FD002 and FD003, the RMSE results given by our proposal noticeably advance the state-of-the-art. In terms of s-score, while the proposed method does not clearly outperform state-of-the-art algorithms, its test values are comparable to the best scores in the table. The outstanding performance of our model can be highlighted when we consider the RMSE results rather than the s-score results. Overall, although our RUL prediction model tends to provide somewhat "optimistic" predictions (see Section 5.2), it is a reliable RUL prediction tool considering its outstanding prediction accuracy.

5.4.2 MOO of neural architectures

ELM

Our experiments aim to achieve the MOO of the ELM on the N-CMAPSS dataset by applying the MOO algorithm introduced in Section 4.2 where the multiple objectives are the RUL prediction error and the number of trainable parameters, and we numerically evaluate the quality of the discovered solutions by comparing them with the different methods explained below, in terms of the test RMSE on D_{test} of the N-CMAPSS and the number of trainable parameters.

To demonstrate the advantage of the MOO of the ELM, we consider two different scenarios in terms of SOO, which we refer to respectively as SOO-ELM(1) and SOO-ELM(2). The ELMs obtained by the SOO approaches are

then compared with the proposed method. In the first case, SOO-ELM(1), we find the solutions by solving an SOO optimization problem where the objective function is validation RMSE, i.e., it aims at minimizing simply the RUL prediction error. The limitation of this approach is that the size of the discovered ELM tends to be very large in order to decrease the validation RMSE. In other words, n_{tanh} and n_{sigm} specified in Table 4.1 tend to converge to their upper bounds throughout the evolutionary process.

To overcome this limitation, we consider a second SOO approach, SOO-ELM(2), in which the fitness formulation includes the number of trainable parameters of the ELM. This conventional approach aims to solve the multi-objective problem as a single-objective problem by considering scalarization which combines the two different objectives, the validation RMSE and the number of trainable parameters (denoted by $RMSE_{val}$ and L respectively) in our case, into one linear function; here, the scalarization function is a weighted sum of the objectives: $RMSE_{val}+\tau L$, where τ is a constant weight. This way, we can prevent the survival of unnecessarily large ELMs, which are the solutions of the first approach, by penalizing their fitness with τL.

Minimizing $RMSE_{val}$ and L are conflicting objectives in the architecture search of the ELMs. While the SOO-ELM(2) approach discussed above somehow goes in the direction of compromising those two objectives, the best model still largely depends on a human decision, since it depends on how the value of τ is parametrized. To tackle this limitation, as discussed in Section 4.2, we propose to use a MOO algorithm, NSGA-II, to search explicitly for a set of trade-off ELMs. We refer to this method as MOO-ELM.

In all three approaches, the fitness of each individual is calculated by generating an ELM (the phenotype) associated to the corresponding genotype, i.e., a vector containing the three parameters introduced in Section 4.1.1. In particular, the validation RMSE of its phenotype is evaluated on D_v, after training it on D_w.

Regarding the experiments of MOO-ELM, we execute 10 independent runs with different random seeds to improve the reliability of the results from the GA-based methods by considering different initial population. n_{pop} and n_{gen} are set to 28 and 30 respectively. Note that, as discussed in the paper[14], we observed a gradual improvement across the generations of MOO-ELM and verified that the algorithm explores the search space enough within 30 generations. For each MOO-ELM run, the evolutionary search returns a subset of the trade-off solutions that have the top dominance level (i.e., the method returns a Pareto front) after the fixed number of generations.

To compare the results obtained by the different methods under study, we aggregate the 10 independent runs in the following way: in the case of the two SOO-ELM approaches, each run returns a single solution that has the best fitness during evolution. The aggregation is then simply the mean of the test RMSE of the 10 best individuals. On the other hand, each run of NSGA-II returns a number of solutions on the final Pareto front. In our case, we collected 417 non-dominated solutions across the 10 runs. For the sake of comparison, instead of using all of them, we select a fraction of the solutions based on their density in the fitness space, as described in Figure 5.6. Specifically, for further analysis, the fitness space is discretized in 20×20 bins. The bin highlighted in yellow is the one with the highest density of solutions. When we do not have any preference for a certain objective, this strategy can be used to derive a subset of the solutions which are implicitly "preferred" by the MOO algorithm. As shown in Figure 5.6, we first place all the solutions from the 10 runs in the fitness space, which is discretized in equally-spaced bins. The density of the solutions can then be measured by counting the number of solutions lying in each bin. As a result, we choose the 28 solutions from the bin with the highest density and use the average of their test RMSE as the final result for MOO-ELM, shown in Table 5.9.

In addition, we compare the solutions obtained by the above three GA-

based approaches with BPNNs: a MLP and a CNN whose architectures were manually designed in[98]. The architecture of the considered MLP has four hidden layers, and the CNN is made up of three convolutional layers followed by a fully connected layer. Further details about the architectures of the BPNNs and their training are specified in our paper[14]. Of note, a simple feed-forward neural network (the MLP) and a 1-D CNN are still used as state-of-the-art neural networks on the N-CMAPSS dataset, considering the great amount of data (in the order of millions of samples) obtained from real flight conditions.

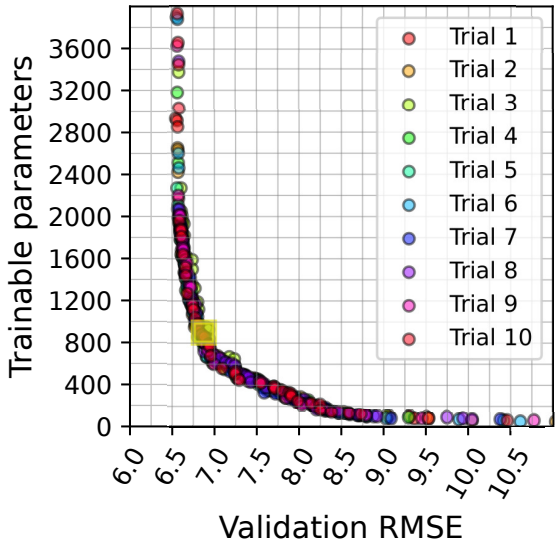


Figure 5.6: Trade-off between validation RMSE and number of trainable parameters at the last generation for the 10 independent runs of the proposed MOO-ELM approach (aggregate results across runs by discretizing fitness space).

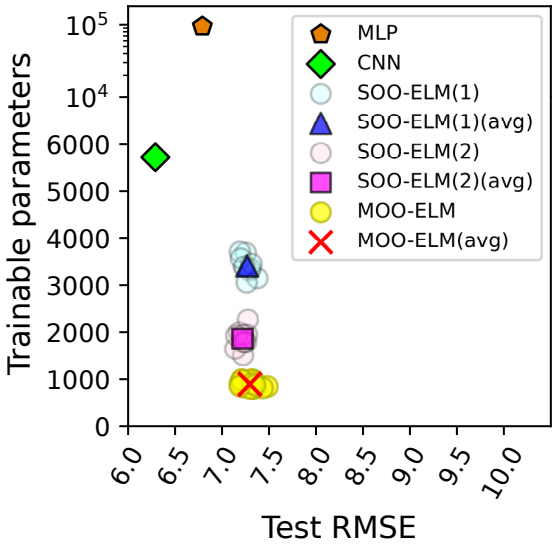


Figure 5.7: Trade-off between test RMSE and number of trainable parameters for the methods considered in the experimentation. For the results of ELM we report the result of each of the 10 available runs, and their average.

The comparative results of all the considered methods are presented in Table 5.9 and visualized in Figure 5.7. It can be seen that MOO-ELM achieves comparable results to state-of-the-art MLP and CNN models designed by

human experts[98]. For MLP and CNN, we report only one value related to one single run (since their computations are deterministic). Note that the impact of the random initialization of the weights in NNs is not examined in our experiments (i.e., we assume that each architecture yields deterministic performance after its training), while we take into account 10 different random seeds for our evolutionary search. Each run of the GA with a different random seed produces indeed a different set of solutions. Therefore, having multiple runs of the proposed evolutionary algorithm allows us to verify its robustness over different random seeds.

For the proposed methods, SOO-ELM(1), SOO-ELM(2), and MOO-ELM, we report mean ± SD obtained across 10 independent runs. The boldface indicates the best result in each column. As mentioned earlier, within the scope of data-driven methods, those deep networks offer indeed state-of-the-art RUL predictions (in terms of test RMSE) on the N-CMAPSS dataset. However, the MLP contains a huge amount of trainable parameters throughout four stacked layers, and training those parameters with an iterative approach requires more than 18 minutes. The CNN shows better prediction accuracy with a lower number of parameters by leveraging parameter sharing, but the training time of this DL architecture is almost twice as big as that of the MLP. Note that the test RMSE values of the BPNNs in Table 5.9 are different from those reported in[98], which are based on an early version of DS02 that has a lower noise level on the sensor readings and a sampling rate of 0.1Hz (instead of 1Hz).

On the contrary, all the optimized ELMs have a considerably smaller number of trainable parameters, which reflects a much shorter (up to 2 orders of magnitude) training time. The architecture discovered by SOO-ELM(1) tends to have almost the maximum available number of hidden neurons because it simply uses the validation RMSE as the fitness for its evolutionary search. Yet, in this case, the ELM does not suffer from overfitting and its per-

Table 5.9: Summary of the comparative results analysis of the ELM optimized by the GA with handcrafted BPNNs.

Methods	Architecture	Test RMSE (on D_{test})	Trainable parameters	Training time (s)
MLP [98]	4 hidden layers	6.79	94,701	1,081
CNN [98]	3 convolutional layers	**6.29**	5,722	1,969
SOO-ELM(1) [14]	-	7.27±0.05	3,405±202	337±49
SOO-ELM(2) [14]	-	7.21±0.04	1,859±207	110±22
MOO-ELM [14]	-	7.29±0.07	**898±60**	**55±10**

formance does not get worse even if we increase the number of hidden neurons excessively. In other words, the oversized ELM network can merely achieve a negligible improvement but requires unnecessarily large computational costs for training the redundant parameters. The SOO-ELM(2) approach, on the other hand, penalizes those oversized ELMs by introducing the penalty factor. We set the constant weight τ to 10^{-4}, small enough for the penalty not to dominate the overall fitness. We can see that SOO-ELM(2) successfully prevents the use of the redundant neurons, so that it can preserve the RUL prediction accuracy using only almost half the neurons, w.r.t. the previous method.

Although SOO-ELM(2) uses almost half the neurons and requires less than two minutes of training time, a proper value of τ must be determined by empirical considerations. In contrast, MOO-ELM can overcome this problem using NSGA-II for automatically searching a set of trade-off network architectures without considering a tunable parameter such as the τ used before. Compared to SOO-ELM(2), this method achieves almost the same test RMSE but can further halve the number of trainable parameters. Moreover, it only needs, on average, less than one minute to train the best trade-off networks.

Finally, the comparative results are visualized in Fig. 5.7, which easily

allows us to compare the performance of the different methods in terms of the trade-off between the two conflicting objectives. We can observe that the CNN dominates the MLP. Among the compared algorithms, MOO-ELM obtains the best solutions in terms of the number of trainable parameters (on average, about 900, i.e., less than 16% compared to the CNN). Moreover, compared to the CNN, the models discovered by MOO-ELM have an approximately 97% shorter training time, while their test RMSE is on average only 16% larger.

CELM

The description in this subsection is based on the paper by Mo et al.[15] that extends the previous work[14] in which we tackled the data-driven RUL prediction task by means of the ELM optimization and tested on the N-CMAPSS. In the extended paper, we test the optimized ELMs by our MOO algorithm, namely MOO-ELM, on the CMAPSS (i.e., as one of the new contents, we test the idea presented in the previous work[14] on another dataset, the CMAPSS); in addition, we apply the MOO algorithm to the CELM search space defined in Section 4.1.2, and the found solutions, the optimized CELMs, are evaluated on the CMAPSS by comparing their experimental results to the numerical results reported in the literature using different methods.

As such, the aim of our experiments is to evaluate the optimized ELMs and CELMs discovered by the proposed methods, where MOO-ELM and MOO-CELM denote the optimized ELMs and CELMs respectively. To perform a thorough evaluation, we compare them with traditional BPNNs widely used in the field of RUL prediction not only in terms of the number of trainable parameters but also in terms of RUL prediction performance, the latter being based on the two metrics: RMSE and s-score.

Particularly, for a comparative analysis, we consider three handcrafted networks that have been tested on the CMAPSS dataset: a MLP[44], CNN[44],

and LSTM[48]. The architecture of the MLP comprises one hidden layer of 50 neurons. Regarding CNN, the model consists of two pairs of convolutional layers and pooling layers, followed by a fully-connected layer. The first convolutional layer has 8 filters of size 12, while the following convolutional layer contains 14 filters of size 4. Each pooling layer performs average pooling with size 1×2 to halve the feature length. The feature map is flattened at the end of the last pooling layer and passed to the fully-connected layer of 50 neurons. Both the MLP and the CNN use a sigmoid as an activation function. The LSTM has four hidden layers: two stacked LSTM layers and two fully-connected layers. The number of hidden units in each LSTM is 32, and the following two fully-connected layers contain 8 neurons in each layer.

Regarding the proposed evolutionary search, n_{pop} and n_{gen} are set both to 20; as discussed in the paper[15], we have empirically found that these values allow enough evaluations to observe an improvement of the solution quality across the generations. For each parameter space (i.e., for the ELMs and CELMs) and each of the four CMAPSS sub-datasets, we execute 10 independent runs of the MOO algorithm with different random seeds. The multiple runs are considered to enhance the reliability of the results obtained by the proposed methods based on NSGA-II, and each run of the evolutionary search returns multiple solutions on the final Pareto front. Here, to perform a comparative analysis with the other methods we take a subset of the solutions from the 10 independent runs by using the aggregation method described in Section 5.4.2. After the aggregation, our MOO-ELM comprises 15, 31, 20 and 30 solutions in the four sub-datasets respectively. In the case of MOO-CELM, the numbers of selected solutions are 6, 4, 6 and 4. In each experiment, we calculate the test RMSE, the s-score, and the number of trainable parameters, for each of the available solutions. Their averages are then computed as the final results which are compared with the other methods employing the BPNNs.

Table 5.10: Summary of results analysis of MOO-ELM and MOO-CELM on CMAPSS, compared with handcrafted BPNNs in terms of test RMSE and number of trainable parameters.

Method	RMSE				Trainable parameters
	FD001	FD002	FD003	FD004	
MLP [44]	37.36±0.00	80.03±0.00	37.39±0.00	77.37±0.00	801
CNN [44]	18.45±0.00	30.29±0.00	19.82±0.00	29.16±0.00	6,815
LSTM [48]	**16.14±0.00**	**24.49±0.00**	**16.18±0.00**	**28.17±0.00**	14,681
MOO-ELM [14]	18.93±0.19	30.46±0.12	20.56±0.15	31.70±0.19	**326**
MOO-CELM [15]	16.54±0.57	39.98±0.35	17.97±0.80	42.62±0.78	751

Table 5.11: Summary of results analysis of MOO-ELM and MOO-CELM on CMAPSS, compared with handcrafted BPNNs in terms of s-score and number of trainable parameters.

Method	s-score $\times$ (10^3)				Trainable parameters
	FD001	FD002	FD003	FD004	
MLP [44]	18.00±0.00	7800.00±0.00	17.40±0.00	5620.00±0.00	801
CNN [44]	1.29±0.00	13.60±0.00	1.60±0.00	7.89±0.00	6,815
LSTM [48]	**0.34±0.00**	**4.45±0.00**	0.85±0.00	**5.55±0.00**	14,681
MOO-ELM [14]	1.12±0.08	14.31±0.41	2.12±0.16	10.63±0.19	**326**
MOO-CELM [15]	0.46±0.07	64.62±1.90	**0.64±0.15**	47.12±0.98	751

The comparative results of all the considered methods are presented in Table 5.10 and Table 5.11. We report the values in terms of mean ± SD over 10 independent runs for our methods, namely, MOO-ELM and MOO-CELM, Note that the SD of the trainable parameters is neglected because it is relatively small. For the remaining methods, we report only one solution related to one single run since their computations are deterministic.

In terms of test RMSE, our methods are much better than the MLP, since we can obtain lower RMSE values with a smaller number of trainable parameters. Although both the proposed methods and the MLP use hundreds of parameters, our methods are considerably better in terms of computational

cost and training time because we apply an extremely fast ELM learning algorithm while the MLP is trained by BP, which is relatively slow and expensive. The CNN achieves a much better test RMSE but has an even larger number of trainable parameters, compared to the MLP. Nevertheless, the test results of the proposed methods are still fairly comparable to those obtained by the CNN in terms of test RMSE, while our methods achieve these results by using a much smaller number of trainable parameters as well as a much faster learning algorithm. In particular, in terms of test RMSE, the results of MOO-ELM are slightly worse but very close to the results of the CNN for all four sub-datasets. MOO-CELM outperforms the CNN in terms of test RMSE on FD001 and FD003, but it does not provide as good results on the remaining sub-datasets. This implies that MOO-ELM can achieve a sufficient and stable performance on all the datasets including FD002 and FD004 that contain the data of six working conditions, while MOO-CELM is only advantageous for the RUL prediction task on less complicated datasets such as FD001 and FD003 collected under only one condition.

Additionally, we compare our results to the LSTM; as shown in Table 5.10, it is the best method in terms of test RMSE, but the number of trainable parameters in the LSTM is more than two orders of magnitude larger. In fact, the solutions given by MOO-CELM not only have a clear advantage in terms of number of trainable parameters but also can produce low prediction errors, which are comparable to those achieved by the LSTM on the two less challenging datasets.

Most of this analysis on the test RMSE is also valid for the s-score results summarized in Table 5.11. When we look at the score values in this table, those given by MOO-ELM are close to the results of the CNN. Moreover, MOO-CELM achieves the best score among the compared methods on FD003, as well as a good score (close to the score of the LSTM) on FD001.

Finally, the comparative results are visualized in Figure 5.8, which easily

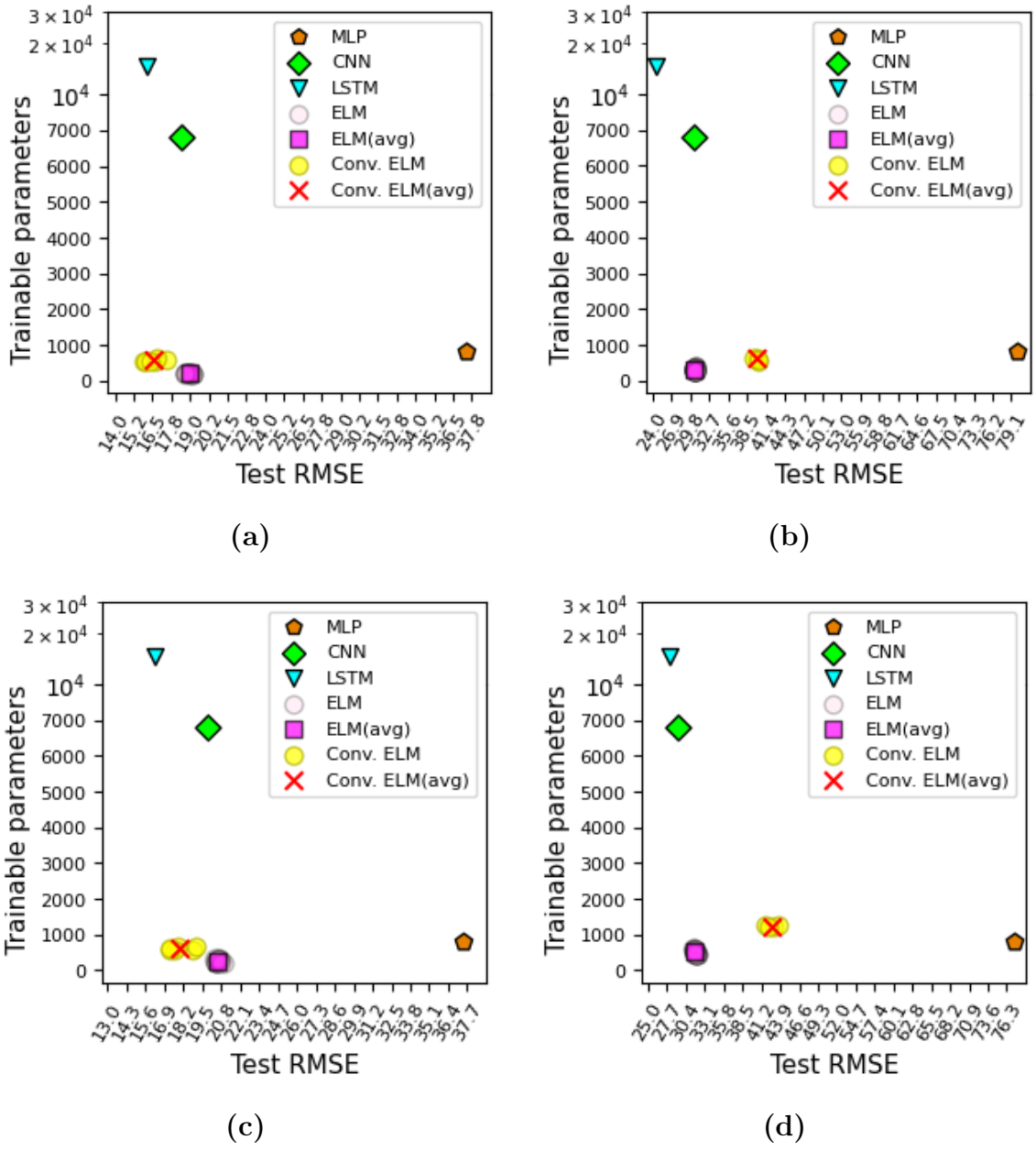


Figure 5.8: Trade-off between test RMSE and number of trainable parameters for the methods considered in the experimentation. ELM(avg) and Conv. ELM(avg) correspond to the results by MOO-ELM and MOO-CELM, respectively, reported in Table 5.10: (a) FD001 dataset; (b) FD002 dataset; (c) FD003 dataset; (d) FD004 dataset.

allows comparing the performance of the different methods in terms of trade-off between the two conflicting objectives; since the analysis of the score

results is not much different from that carried out on the RMSE results, in the figure we only illustrate the results w.r.t. the test RMSE, previously reported in Table 5.10.

We can observe that none of the BPNNs dominates the others. Among the BPNNs, the MLP uses a lower number of trainable parameters, while the LSTM offers the best performance in terms of test RMSE. The CNN is placed between the MLP and the LSTM for both objectives.

When we compare the two proposed methods with each other, using MOO-CELM instead of MOO-ELM improves the RUL prediction for less complicated data, as shown in Figure 5.8 (a) and (c), although the number of trainable parameters slightly increases. In contrast, we find that introducing randomly generated convolutional filters can disturb the RUL prediction made by the ELMs on FD002 and FD004.

The proposed methods, MOO-ELM and MOO-CELM, dominate the MLP for all the datasets except for FD004. In Figure 5.8 (d), the number of trainable parameters of MOO-CELM is slightly larger than that of the MLP, but our method is still better in terms of computational cost because it uses ELM, which is a much more efficient training algorithm compared to BP. In the two less challenging datasets, FD001 and FD003, the solutions discovered by MOO-CELM dominate the CNN. The results given by MOO-ELM show a comparable prediction performance while using a significantly lower number of trainable parameters, compared to the CNN. Although our proposed methods cannot outperform the LSTM in terms of test RMSE, they still can be good RUL prediction tools considering this much smaller number of trainable parameters and the advantages of ELM training.

1-D CNN

In the rest of this section, driven by the motivation clarified in Section 4.1.3, we consider a 1-D CNN as our backbone network. More specifically, the

MOO approach proposed in Chapter 4 is proposed to develop RUL prediction tools used for some industrial contexts that only allow access to resource-constrained HW devices. As such, it is reasonable to apply our MOO to the 1-D CNN which shows good performance despite its simple structure that does not rely on recurrence in the data-driven RUL prediction tasks. Whereas the speed-up techniques are not necessary for the optimization of the ELM as discussed above (because of its extremely fast training explained in Section 4.1.1), those are needed for accelerating the evolutionary NAS that explores the combinatorial parameter space of the 1-D CNN described in Table 4.3. The evolutionary NAS is computationally expensive because each individual (i.e., candidate 1-D CNN architecture) should tune its parameters iteratively with gradient-based computations until convergence, before being evaluated on the validation data. The goal is to shorten the lengthy training process (and, as a consequence, the total time of the evolutionary search). Particularly, two of the fitness prediction strategies explained in Section 3.3, learning curve extrapolation (Section 3.3.2) and zero-cost proxy (Section 3.3.3), are selected as the speed-up techniques for accelerating our multi-objective GA; the former facilitates the evaluation of each network with training for a reduced number of epochs, and the latter improves the efficiency of evaluations by not training at all.

Our experiments aim to verify that applying the speed-up techniques can meaningfully reduce the runtime of our multi-objective evolutionary search without compromising the quality of the solutions. In detail, first, we generate 20 individuals (i.e., 1-D CNNs) randomly. Then, the multi-objective evolutionary process starts from the initial population. In our experiments, we set the maximum number of epochs n_m to 30, based on our previous works[12,14,68]. 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. To perform a comparative analysis, we consider 5 different configurations differentiated by the way they define 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 Figure 5.9,

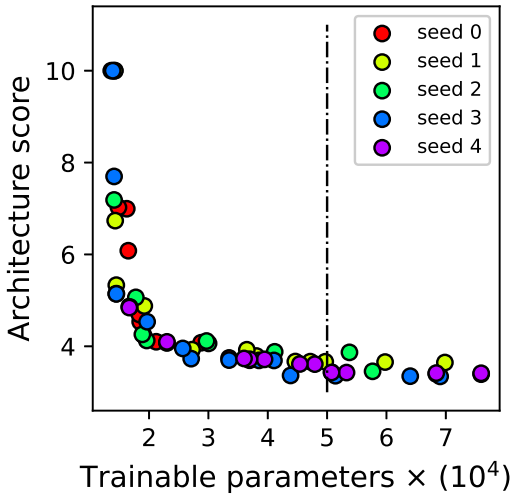


Figure 5.9: Architecture score vs. number of trainable parameters on 100 randomly generated networks (20 for each seed). The dash-dotted line indicates the decision threshold.

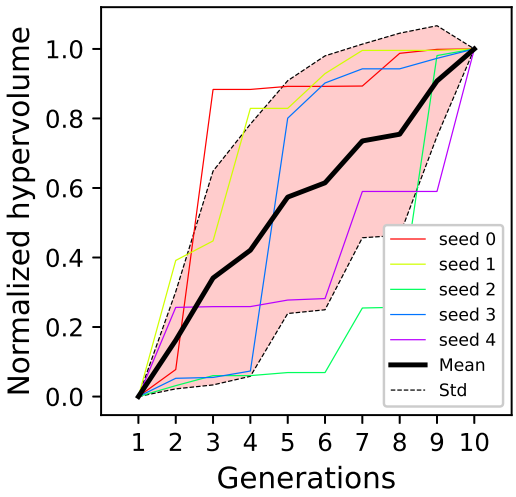


Figure 5.10: Normalized validation HV across generations (mean ± standard deviation across 5 independent runs) for the evolutionary runs of the proposed NAS for MOO.

Table 5.12: Summary of the comparative results analysis of the 1-D CNNs optimized by the MOO for 5 different NAS configurations w.r.t. $fitness_{RMSE}$.

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

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 negligible, 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 a fitness value (from the GA perspective), we 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 3.3.2. 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 Figure 5.10 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 as 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. Figure 5.11 shows the results of our experiments. In the figure, we present the HV values w.r.t. 5 different configurations: 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 the reference point (13, 13) which is a reference point in HV calculation. Each figure shows the solutions found in 5 independent runs. The results of the handcrafted CNN, used as a baseline, are taken from [98].

Table 5.12 describes the summary of the comparative analysis; the HV is an avg.±std. of the values in Figure 5.11 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. 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 distinguish 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 that 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 reduces 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.

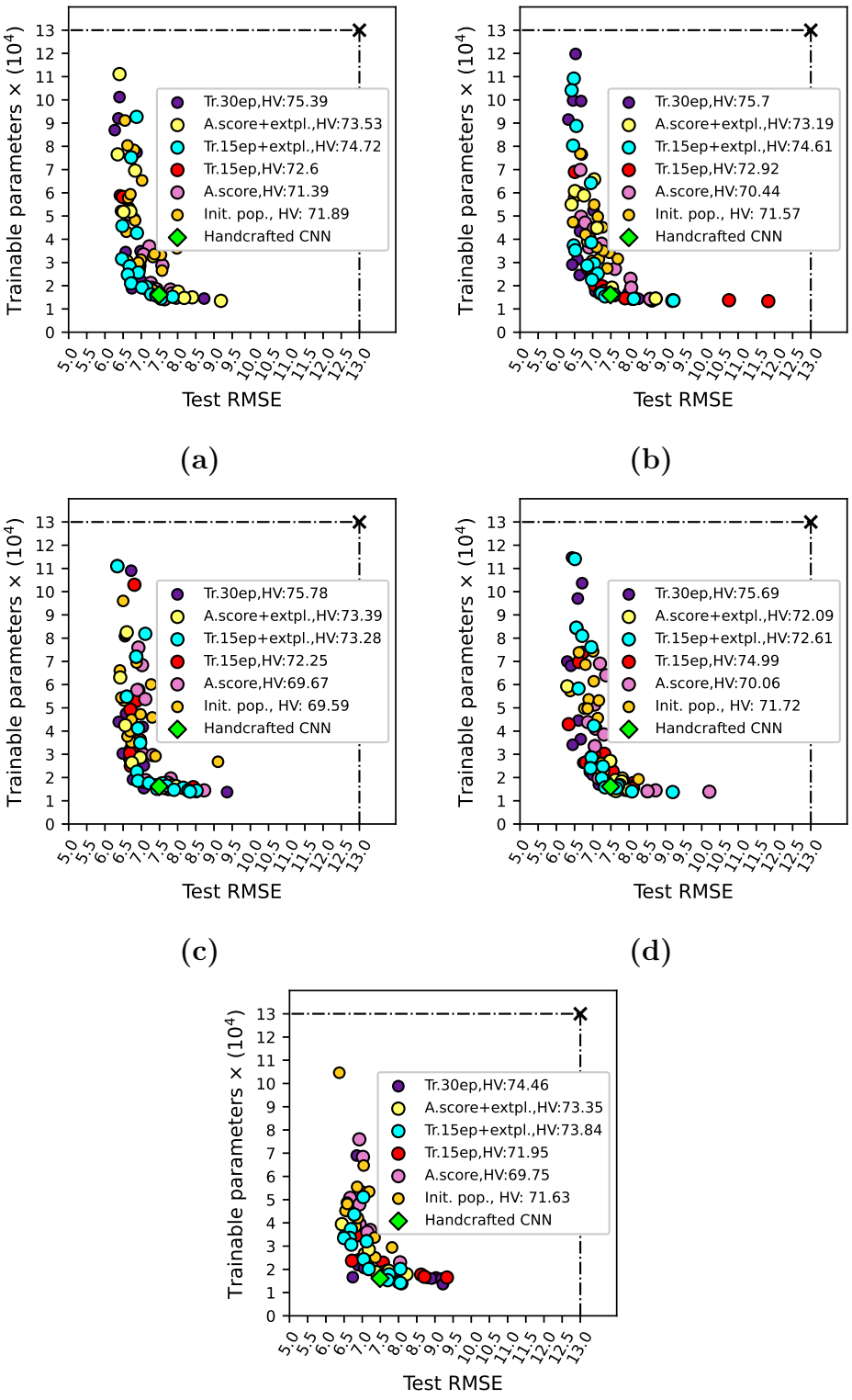


Figure 5.11: Pareto front for 5 different 1-D CNN NAS configurations w.r.t. $fitness_{RMSE}$.

Chapter 6

Conclusion

Our works introduce NAS algorithms that employ evolutionary computation to explore the combinatorial parameter space for different neural architectures to be used for RUL predictions. In Chapter 3, they are developed to solve the SOO problem of finding the best architectures of multi-head CNN-LSTM and Transformers in terms of RUL prediction accuracy; the works presented in this chapter are based on the papers[12,13]. Considering industrial contexts that normally seek to save cost by minimizing access to expensive computing infrastructures, in Chapter 4, we applied the NAS to solve the MOO problem of finding the optimal architectures of ELM and CELM and 1-D CNN for achieving a trade-off between two competing objectives: the RUL prediction error and the number of trainable parameters; the works described in this chapter are based on the papers[14–16].

In particular, GAs are used to solve optimization problems and the GA-based NAS in this work can not only find high-quality solutions but also provide a fast and efficient search. To speed up the evolutionary search by the GA, several fitness prediction techniques are considered. Early-stopping and learning curve extrapolation offer a reduced number of training epochs for evaluating the fitness of a network. Zero-cost proxy predicts the fitness of a fully-trained network at initialization. The model-based prediction employs

the regression model which works as a surrogate for the fitness observation after training.

In the experiments described in Chapter 5, the comparative evaluations are based on the CMAPSS dataset and the N-CMAPSS dataset which serve as a benchmark enabling a better comparison of different algorithms. In the experiments on multi-head CNN-LSTM architecture optimization, we verify that the evolutionary NAS can discover better neural architectures in terms of the two metrics, RMSE and s-score, compared to the architecture manually designed by domain experts. From the experimental results of the evolutionary NAS on the Transformers, we can find that the prediction error performance of the solutions found by the proposed GA is overall better than those obtained by a space-filling sampling. Moreover, compared to the current state of the art, the architecture-optimized Transformers provides better results in terms of RMSE and fairly comparable results in terms of s-score. In the case of the MOO of the 1-D CNN, the experimental results on the benchmark show that the GA with the fitness predictors can achieve much better solution quality than randomly generated networks as well as save a considerable amount of the GA runtime. Regarding the optimization of the ELM and CELM, the solutions given by the MOO of both ELM and CELM networks provide fairly comparable results with handcrafted BPNNs such as MLP and CNN models; nevertheless, the number of trainable parameters is significantly smaller and the training time is much shorter.

Overall, our works demonstrate that using the GA-based NAS with fitness prediction techniques or ELM, the proposed evolutionary search of the networks having optimized architecture with regards to both single and multi-objective optimization, can be a useful tool to solve the RUL prediction task. This can contribute to having an effective RUL prediction tool that offers a very precise prediction based on a deep and complex neural architecture and its optimization. The MOO of the relatively simple networks that have prim-

itive structures can be used as a way to generate an efficient RUL prediction tool in an industrial context that requires finding a compromise between the prediction accuracy and the number of parameters, which in turn correlates with memory consumption (which is a crucial aspect e.g., in embedded systems) and computing time.

Papers to appear or published during the PhD

Selected for the thesis

published

Hyunho Mo, Leonardo Lucio Custode, and Giovanni Iacca. Evolutionary neural architecture search for remaining useful life prediction. Applied Soft Computing, 108:107474, 2021: in this journal paper, we propose a GA to optimize the architecture of multi-head CNN LSTM and use the found solutions to predict the RUL of aircraft engines. Note that Chapter 3 is based on this paper and its experimental results are presented in Chapter 5.

Hyunho Mo and Giovanni Iacca. Evolutionary neural architecture search on transformers for remaining useful life prediction. Materials and Manufacturing Processes, pages 1–18, 2023: in this journal paper, we introduce a surrogate-assisted evolutionary NAS to optimize the architecture of the Transformers and use the found solutions to predict the RUL of aircraft engines. Note that Chapter 3 is based on this paper and its experimental results are presented in Chapter 5.

Hyunho Mo and Giovanni Iacca. Multi-objective optimization of extreme learning machine for remaining useful life prediction. In International Conference on the Applications of Evolutionary Computation (Part of EvoStar), pages 191–206. Springer, 2022: in this conference paper, we apply a MOO algorithm to optimize the ELM and use the found solutions to predict the RUL of aircraft engines. Note that Chapter 4 is based on this paper and its experimental results are summarized in Chapter 5.

Hyunho Mo and Giovanni Iacca. Accelerating evolutionary neural architecture search for remaining useful life prediction. In International Conference on Bioinspired Optimization Methods and Their Applications, pages 15–30. Springer, 2022: in this conference paper, we present the architecture optimization of 1-D CNN by leveraging a MOO algorithm in conjunction with speeding up techniques for the evolutionary search and use the found solutions to predict the RUL of aircraft engines. Note that Chapter 4 is based on this paper and its experimental results are described in Chapter 5.

to appear

Hyunho Mo and Giovanni Iacca. Evolutionary optimization of convolutional extreme learning machine for remaining useful life prediction. SN Computer Science, 2023. to appear: in this journal paper, we apply a MOO algorithm to optimize the CELM and use the found solutions to predict the RUL of aircraft engines. Note that Chapter 4 is based on this paper and its experimental results are presented in Chapter 5.

Not selected for the thesis

published

Hyunho Mo, Federico Lucca, Jonni Malacarne, and Giovanni Iacca. Multi-head cnn-lstm with prediction error analysis for remaining useful life prediction. In 2020 27th Conference of Open Innovations Association (FRUCT), pages 164–171. IEEE, 2020: in this conference paper, we manually design a multi-head CNN-LSTM network and use it to predict the RUL of aircraft engines. Note that its experimental results are included in Chapter 5 and used to compare and evaluate the results from the selected papers.

Leonardo Lucio Custode, Hyunho Mo, and Giovanni Iacca. Neuroevolution of spiking neural p systems. In Applications of Evolutionary Computation: 25th European Conference, EvoApplications 2022, Held as Part of EvoStar 2022, Madrid, Spain, April 20–22, 2022, Proceedings, pages 435–451. Springer, 2022: in this conference paper, we introduce the NEAT neuroevolutionary algorithm to evolve Spiking Neural P systems.

Leonardo Lucio Custode, Hyunho Mo, Andrea Ferigo, and Giovanni Iacca. Evolutionary optimization of spiking neural p systems for remaining useful life prediction. Algorithms, 15(3):98, 2022: in this journal paper, we develop Spiking Neural P systems by means of a neuro-evolutionary algorithm to predict the RUL of aircraft engines.

Bibliography

[1] Dean K Frederick, Jonathan A DeCastro, and Jonathan S Litt. User's guide for the commercial modular aero-propulsion system simulation (c-mapss). *NASA Technical Manuscript*, 2007–215026, 01 2007.

[2] Euclides da Conceição Pereira Batalha. *Aircraft Engines Maintenance Costs and Reliability: An Appraisal of the Decision Process to Remove an Engine for a Shop Visit Aiming at Minimum Maintenance Unit Cost*. PhD thesis, Estatística e Gestão de Informação, Universidade Nova de Lisboa, 2012.

[3] Tiedo Tinga. Predicting critical failures using physics of failure: opportunities and challenges. In *AVT-356 Research Symposium on Physics of Failure for Military Platform Critical Subsystems*, pages 1–13. NATO Science & Technology Organization, 2021.

[4] Olga Fink. Data-driven intelligent predictive maintenance of industrial assets. *Women in Industrial and Systems Engineering: Key Advances and Perspectives on Emerging Topics*, pages 589–605, 2020.

[5] Enrico Zio. Prognostics and health management (phm): Where are we and where do we (need to) go in theory and practice. *Reliability Engineering & System Safety*, 218:108119, 2022.

[6] Olga Fink, Qin Wang, Markus Svensen, Pierre Dersin, Wan-Jui Lee, and Melanie Ducoffe. Potential, challenges and future directions for

deep learning in prognostics and health management applications. *Engineering Applications of Artificial Intelligence*, 92:103678, 2020.

[7] George J Vachtsevanos and George J Vachtsevanos. *Intelligent fault diagnosis and prognosis for engineering systems*, volume 456. Wiley Online Library, 2006.

[8] Yuqiao Liu, Yanan Sun, Bing Xue, Mengjie Zhang, Gary G Yen, and Kay Chen Tan. A survey on evolutionary neural architecture search. *IEEE transactions on neural networks and learning systems*, 2021.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[10] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: a new learning scheme of feedforward neural networks. In *2004 IEEE international joint conference on neural networks (IEEE Cat. No. 04CH37541)*, volume 2, pages 985–990. Ieee, 2004.

[11] Michel M Dos Santos, Abel G da Silva Filho, and Wellington P dos Santos. Deep convolutional extreme learning machines: Filters combination and error model validation. *Neurocomputing*, 329:359–369, 2019.

[12] Hyunho Mo, Leonardo Lucio Custode, and Giovanni Iacca. Evolutionary neural architecture search for remaining useful life prediction. *Applied Soft Computing*, 108:107474, 2021.

[13] Hyunho Mo and Giovanni Iacca. Evolutionary neural architecture search on transformers for rul prediction. *Materials and Manufacturing Processes*, pages 1–18, 2023.

[14] Hyunho Mo and Giovanni Iacca. Multi-objective optimization of extreme learning machine for remaining useful life prediction. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pages 191–206. Springer, 2022.

[15] Hyunho Mo and Giovanni Iacca. Evolutionary optimization of convolutional extreme learning machine for remaining useful life prediction. *SN Computer Science*, 2023. to appear.

[16] Hyunho Mo and Giovanni Iacca. Accelerating evolutionary neural architecture search for remaining useful life prediction. In *International Conference on Bioinspired Optimization Methods and Their Applications*, pages 15–30. Springer, 2022.

[17] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[18] Abhinav Saxena, Kai Goebel, Don Simon, and Neil Eklund. Damage propagation modeling for aircraft engine run-to-failure simulation. In *International Conference on Prognostics and Health Management (ICPHM)*, pages 1–9. IEEE, 2008.

[19] Manuel Arias Chao, Chetan Kulkarni, Kai Goebel, and Olga Fink. Aircraft engine run-to-failure dataset under real flight conditions for prognostics and diagnostics. *Data*, 6:5, 2021.

[20] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet clas-

sification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[22] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[24] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[25] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017.

[26] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *International Conference on Machine Learning*, pages 2902–2911. PMLR, 2017.

[27] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[28] Zefeng Chen, Yuren Zhou, and Zhengxin Huang. Auto-creation of effective neural network architecture by evolutionary algorithm and resnet

for image classification. In *2019 IEEE international conference on systems, man and cybernetics (SMC)*, pages 3895–3900. IEEE, 2019.

[29] Yanan Sun, Bing Xue, Mengjie Zhang, Gary G Yen, and Jiancheng Lv. Automatically designing cnn architectures using the genetic algorithm for image classification. *IEEE transactions on cybernetics*, 50(9):3840–3854, 2020.

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

[31] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[32] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[33] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018. arXiv:1810.04805.

[34] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45. Association for Computational Linguistics, 2020.

[35] Linhao Dong, Shuang Xu, and Bo Xu. Speech-transformer: a no-recurrence sequence-to-sequence model for speech recognition. In *Inter-*

national Conference on Acoustics, Speech and Signal Processing, pages 5884–5888. IEEE, 2018.

[36] Nanxin Chen, Shinji Watanabe, Jesús Villalba, Piotr Żelasko, and Najim Dehak. Non-autoregressive transformer for speech recognition. *IEEE Signal Process Lett.*, 28:121–125, 2020.

[37] Thomas Watts, Bing Xue, and Mengjie Zhang. Blocky net: A new neuroevolution method. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 586–593. IEEE, 2019.

[38] Filipe Assunção, Nuno Lourenço, Penousal Machado, and Bernardete Ribeiro. Fast denser: Efficient deep neuroevolution. In *Genetic Programming: 22nd European Conference, EuroGP 2019, Held as Part of EvoStar 2019, Leipzig, Germany, April 24–26, 2019, Proceedings 22*, pages 197–212. Springer, 2019.

[39] Seyed Mohammad Jafar Jalali, Sajad Ahmadian, Md Kislu Noman, Abbas Khosravi, Syed Mohammed Shamsul Islam, Fei Wang, and João PS Catalão. Novel uncertainty-aware deep neuroevolution algorithm to quantify tidal forecasting. *IEEE Transactions on Industry Applications*, 58(3):3324–3332, 2022.

[40] Evgenia Papavasileiou, Jan Cornelis, and Bart Jansen. A systematic literature review of the successors of "neuroevolution of augmenting topologies". *Evolutionary Computation*, 29(1):1–73, 2021.

[41] Tomohiro Tanaka, Takafumi Moriya, Takahiro Shinozaki, Shinji Watanabe, Takaaki Hori, and Kevin Duh. Automated structure discovery and parameter tuning of neural network language model based on evolution strategy. In *2016 ieee spoken language technology workshop (slt)*, pages 665–671. IEEE, 2016.

[42] Tomohiro Tanaka, Takahiro Shinozaki, Shinji Watanabe, and Takaaki Hori. Evolution strategy based neural network optimization and lstm language model for robust speech recognition. In *CHiME 2016 workshop*, pages 32–35, 2016.

[43] Takahiro Shinozaki and Shinji Watanabe. Structure discovery of deep neural network based on evolutionary algorithms. In *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 4979–4983. IEEE, 2015.

[44] Giduthuri Sateesh Babu, Peilin Zhao, and Xiao-Li Li. Deep convolutional neural network based regression approach for estimation of remaining useful life. In *International Conference on Database Systems for Advanced Applications*, pages 214–228. Springer, 2016.

[45] Luyang Jing, Ming Zhao, Pin Li, and Xiaoqiang Xu. A convolutional neural network based feature learning and fault diagnosis method for the condition monitoring of gearbox. *Measurement*, 111:1–10, 2017.

[46] Xiang Li, Wei Zhang, and Qian Ding. Cross-domain fault diagnosis of rolling element bearings using deep generative neural networks. *IEEE Transactions on Industrial Electronics*, 66(7):5525–5534, 2018.

[47] Qin Wang, Gabriel Michau, and Olga Fink. Domain adaptive transfer learning for fault diagnosis. In *2019 Prognostics and System Health Management Conference (PHM-Paris)*, pages 279–285. IEEE, 2019.

[48] Shuai Zheng, Kosta Ristovski, Ahmed Farahat, and Chetan Gupta. Long short-term memory network for remaining useful life estimation. In *2017 IEEE international conference on prognostics and health management (ICPHM)*, pages 88–95. IEEE, 2017.

[49] Yuting Wu, Mei Yuan, Shaopeng Dong, Li Lin, and Yingqi Liu. Remaining useful life estimation of engineered systems using vanilla lstm neural networks. *Neurocomputing*, 275:167–179, 2018.

[50] Ahmed Elsheikh, Soumaya Yacout, and Mohamed-Salah Ouali. Bidirectional handshaking lstm for remaining useful life prediction. *Neurocomputing*, 323:148–156, 2019.

[51] Kyungnam Park, Yohwan Choi, Won Jae Choi, Hee-Yeon Ryu, and Hongseok Kim. Lstm-based battery remaining useful life prediction with multi-channel charging profiles. *Ieee Access*, 8:20786–20798, 2020.

[52] Kwangsuk Lee, Jae-Kyeong Kim, Jaehyong Kim, Kyeon Hur, and Hagbae Kim. Cnn and gru combination scheme for bearing anomaly detection in rotating machinery health monitoring. In *2018 1st IEEE International conference on knowledge innovation and invention (ICKII)*, pages 102–105. IEEE, 2018.

[53] Honghu Pan, Xingxi He, Sai Tang, and Fanming Meng. An improved bearing fault diagnosis method using one-dimensional cnn and lstm. *Strojniski Vestnik/Journal of Mechanical Engineering*, 64, 2018.

[54] Ling Zheng, Wenhao Xue, Fei Chen, Pengtian Guo, Jiaqi Chen, Biying Chen, and Hongbiao Gao. A fault prediction of equipment based on cnn-lstm network. In *2019 IEEE international conference on energy internet (ICEI)*, pages 537–541. IEEE, 2019.

[55] Xiaohan Chen, Beike Zhang, and Dong Gao. Bearing fault diagnosis base on multi-scale cnn and lstm model. *Journal of Intelligent Manufacturing*, 32(4):971–987, 2021.

[56] Jialin Li, Xueyi Li, and David He. A directed acyclic graph network

combined with cnn and lstm for remaining useful life prediction. *IEEE Access*, 7:75464–75475, 2019.

[57] Lei Ren, Jiabao Dong, Xiaokang Wang, Zihao Meng, Li Zhao, and M Jamal Deen. A data-driven auto-cnn-lstm prediction model for lithium-ion battery remaining useful life. *IEEE Transactions on Industrial Informatics*, 17(5):3478–3487, 2020.

[58] Wennian Yu, II Yong Kim, and Chris Mechefske. An improved similarity-based prognostic algorithm for rul estimation using an rnn autoencoder scheme. *Reliab. Eng. Syst. Safe.*, 199:106926, 2020.

[59] Hui Liu, Zhenyu Liu, Weiqiang Jia, and Xianke Lin. Remaining useful life prediction using a novel feature-attention-based end-to-end approach. *IEEE Trans. Ind. Inf.*, 17(2):1197–1207, 2020.

[60] Yuanhang Chen, Gaoliang Peng, Zhiyu Zhu, and Sijue Li. A novel deep learning method based on attention mechanism for bearing remaining useful life prediction. *Applied Soft Computing*, 86:105919, 2020.

[61] Lu Liu, Xiao Song, Kai Chen, Baocun Hou, Xudong Chai, and Huansheng Ning. An enhanced encoder–decoder framework for bearing remaining useful life prediction. *Measurement*, 170:108753, 2021.

[62] Pedro Lara-Benítez, Manuel Carranza-García, and José C Riquelme. An experimental review on deep learning architectures for time series forecasting. *International Journal of Neural Systems*, 31(03):2130001, 2021.

[63] José F Torres, Dalil Hadjout, Abderrazak Sebaa, Francisco Martínez-Álvarez, and Alicia Troncoso. Deep learning for time series forecasting: a survey. *Big Data*, 9(1):3–21, 2021.

[64] Zimeng Lyu, Shuchita Patwardhan, David Stadem, James Langfeld, Steve Benson, Seth Thoelke, and Travis Desell. Neuroevolution of recurrent neural networks for time series forecasting of coal-fired power plant operating parameters. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1735–1743, 2021.

[65] Paulo Cortez, Pedro José Pereira, and Rui Mendes. Multi-step time series prediction intervals using neuroevolution. *Neural Computing and Applications*, 32(13):8939–8953, 2020.

[66] Youdao Wang, Yifan Zhao, and Sri Addepalli. Remaining useful life prediction using deep learning approaches: A review. *Procedia manufacturing*, 49:81–88, 2020.

[67] Mikel Canizo, Isaac Triguero, Angel Conde, and Enrique Onieva. Multi-head cnn–rnn for multi-time series anomaly detection: An industrial case study. *Neurocomputing*, 363:246–260, 2019.

[68] Hyunho Mo, Federico Lucca, Jonni Malacarne, and Giovanni Iacca. Multi-head cnn-lstm with prediction error analysis for remaining useful life prediction. In *2020 27th Conference of Open Innovations Association (FRUCT)*, pages 164–171. IEEE, 2020.

[69] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale, 2020. arXiv:2010.11929.

[70] Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. Transformers in vision: A survey. *Comput. Surv.*, 54(10s):1–41, 2022.

[71] Jiehui Xu, Haixu Wu, Jianmin Wang, and Mingsheng Long. Anomaly transformer: Time series anomaly detection with association discrepancy, 2021. arXiv:2110.02642.

[72] Shiyang Li, Xiaoyong Jin, Yao Xuan, Xiyou Zhou, Wenhu Chen, Yu-Xiang Wang, and Xifeng Yan. Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting. *Adv. Neural. Inf. Process. Syst.*, 32, 2019.

[73] George Zerveas, Srideepika Jayaraman, Dhaval Patel, Anuradha Bhamidipaty, and Carsten Eickhoff. A transformer-based framework for multivariate time series representation learning. In *Conference on Knowledge Discovery & Data Mining*, pages 2114–2124. ACM SIGKDD, 2021.

[74] Zhizheng Zhang, Wen Song, and Qiqiang Li. Dual-aspect self-attention based on transformer for remaining useful life prediction. *IEEE Trans. Instrum. Meas.*, 71:1–11, 2022.

[75] Jason Ross Brown, Yiren Zhao, Ilia Shumailov, and Robert D Mullins. Wide attention is the way forward for transformers, 2022. arXiv:2210.00640.

[76] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? *Adv. Neural. Inf. Process. Syst.*, 32, 2019.

[77] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-fourth international joint conference on artificial intelligence*, 2015.

[78] Jorge J Moré. The Levenberg-Marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116. Springer, 1978.

[79] Mohamed S Abdelfattah, Abhinav Mehrotra, Łukasz Dudziak, and Nicholas D Lane. Zero-cost proxies for lightweight nas. *arXiv preprint arXiv:2101.08134*, 2021.

[80] Joe Mellor, Jack Turner, Amos Storkey, and Elliot J Crowley. Neural architecture search without training. In *International Conference on Machine Learning*, pages 7588–7598. PMLR, 2021.

[81] Colin White, Arber Zela, Robin Ru, Yang Liu, and Frank Hutter. How powerful are performance predictors in neural architecture search? *Adv. Neural. Inf. Process. Syst.*, 34:28454–28469, 2021.

[82] Tony Duan, Avati Anand, Daisy Yi Ding, Khanh K Thai, Sanjay Basu, Andrew Ng, and Alejandro Schuler. Ngboost: Natural gradient boosting for probabilistic prediction. In *International Conference on Machine Learning*, pages 2690–2700. PMLR, 2020.

[83] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Ann. Stat.*, pages 1189–1232, 2001.

[84] Michael D McKay, Richard J Beckman, and William J Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 42(1):55–61, 2000.

[85] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. Snip: Single-shot network pruning based on connection sensitivity, 2018. arXiv:1810.02340.

[86] Giovanni Iacca, Rammohan Mallipeddi, Ernesto Mininno, Ferrante Neri, and Pannuthurai Nagaratnam Suganthan. Super-fit and population size reduction in compact differential evolution. In *2011 IEEE workshop on memetic computing (MC)*, pages 1–8. IEEE, 2011.

[87] Agoston E Eiben and Cornelis A Schippers. On evolutionary exploration and exploitation. *Fundamenta Informaticae*, 35(1-4):35–50, 1998.

[88] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM computing surveys (CSUR)*, 45(3):1–33, 2013.

[89] Zhe Yang, Piero Baraldi, and Enrico Zio. A comparison between extreme learning machine and artificial neural network for remaining useful life prediction. In *Prognostics and System Health Management Conference (PHM)*, pages 1–7, 2016.

[90] Segun Popoola, Sanjay Misra, and Prof. Aderemi Atayero. Outdoor path loss predictions based on extreme learning machine. *Wireless Personal Communications*, 99, 2018.

[91] Guang-Bin Huang, Qin-Yu Zhu, Kudo Mao, Chee Siew, P. Saratchandran, and Narasimhan Sundararajan. Can threshold networks be trained directly? *IEEE Transactions on Circuits and Systems II: Express Briefs*, 53:187 – 191, 2006.

[92] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: theory and applications. *Neurocomputing*, 70(1-3):489–501, 2006.

[93] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: a new learning scheme of feedforward neural networks. In *International Joint Conference on Neural Networks (IJCNN)*, volume 2, pages 985–990. IEEE, 2004.

[94] Shan Pang and Xinyi Yang. Deep convolutional extreme learning ma-

chine and its application in handwritten digit classification. *Computational intelligence and neuroscience*, 2016, 2016.

[95] Mingxing Duan, Kenli Li, Canqun Yang, and Keqin Li. A hybrid deep learning CNN–ELM for age and gender classification. *Neurocomputing*, 275:448–461, 2018.

[96] Youngmin Park and Hyun S Yang. Convolutional neural network based on an extreme learning machine for image classification. *Neurocomputing*, 339:66–76, 2019.

[97] Yaoming Cai, Zijia Zhang, Qin Yan, Dongfang Zhang, and Mst Jainab Banu. Densely connected convolutional extreme learning machine for hyperspectral image classification. *Neurocomputing*, 434:21–32, 2021.

[98] Manuel Arias Chao, Chetan Kulkarni, Kai Goebel, and Olga Fink. Fusing physics-based and deep learning models for prognostics. *Reliability Engineering & System Safety*, 217:107961, 2022.

[99] Xiang Li, Qian Ding, and Jian-Qiao Sun. Remaining useful life estimation in prognostics using deep convolution neural networks. *Reliability Engineering & System Safety*, 172:1 – 11, 2018.

[100] Serkan Kiranyaz, Turker Ince, Osama Abdeljaber, Onur Avci, and Moncef Gabbouj. 1-d convolutional neural networks for signal processing applications. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8360–8364. IEEE, 2019.

[101] Abhinav Saxena and Kai Goebel. Turbofan engine degradation simulation data set. *NASA Ames Prognostics Data Repository*, 18, 2008.

[102] Abhinav Saxena, Kai Goebel, Don Simon, and Neil Eklund. Damage propagation modeling for aircraft engine run-to-failure simulation.

In *International Conference on Prognostics and Health Management*, pages 1–9. IEEE, 2008.

[103] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. Deap: Evolutionary algorithms made easy. *J. Mach. Learn. Res.*, 13:2171–2175, 2012.

[104] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. arXiv:1412.6980.

[105] André Listou Ellefsen, Emil Bjørlykhaug, Vilmar Æsøy, Sergey Ushakov, and Houxiang Zhang. Remaining useful life predictions for turbofan engine degradation using semi-supervised deep architecture. *Reliab. Eng. Syst. Safe.*, 183:240–251, 2019.

[106] Jialin Li and David He. A Bayesian optimization AdaBN-DCNN method with self-optimized structure and hyperparameters for domain adaptation remaining useful life prediction. *IEEE Access*, 8:41482–41501, 2020.

[107] Wei Ming Tan and T Hui Teo. Remaining useful life prediction using temporal convolution with attention. *AI*, 2(1):48–70, 2021.

[108] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Stat.*, pages 50–60, 1947.