Journal of Machine Learning Research 25 (2024) 1-33 Submitted 2/23; Revised 4/24; Published 6/24

An Algorithmic Framework for the Optimization of Deep Neural Networks Architectures and Hyperparameters

Julie Keisler JULIE.KEISLER@EDF.FR
EDF Lab Paris-Saclay
Bd Gaspard Monge, 91120 Palaiseau
University of Lille & INRIA
170 Av. de Bretagne, 59000 Lille

El-Ghazali Talbi EL-GHAZALI.TALBI@UNIV-LILLE.FR
University of Lille & INRIA
170 Av. de Bretagne, 59000 Lille

Sandra Claudel SANDRA.CLAUDEL@EDF.FR
EDF Lab Paris-Saclay
Bd Gaspard Monge, 91120 Palaiseau

Gilles Cabriel GILLES.CABRIEL@EDF.FR
EDF Lab Paris-Saclay
Bd Gaspard Monge, 91120 Palaiseau

Editor: Ruslan Salakhutdinov

Abstract

In this paper, we propose DRAGON (for DiRected Acyclic Graph OptimizatioN), an algorithmic framework to automatically generate efficient deep neural networks architectures and optimize their associated hyperparameters. The framework is based on evolving Directed Acyclic Graphs (DAGs), defining a more flexible search space than the existing ones in the literature. It allows mixtures of different classical operations: convolutions, recurrences and dense layers, but also more newfangled operations such as self-attention. Based on this search space we propose neighbourhood and evolution search operators to optimize both the architecture and hyper-parameters of our networks. These search operators can be used with any metaheuristic capable of handling mixed search spaces. We tested our algorithmic framework with an asynchronous evolutionary algorithm on a time series forecasting benchmark. The results demonstrate that DRAGON outperforms state-of-the-art handcrafted models and AutoML techniques for time series forecasting on numerous datasets. DRAGON has been implemented as a python open-source package[1].

Keywords: neural architecture search, hyperparameters optimization, metaheuristics, evolutionary algorithm, time series forecasting

1. Introduction

With the recent successes of deep learning in many research fields, deep neural networks (DNN) optimization stimulates the growing interest of the scientific community (Talbi, 2021). While each new learning task requires the handcrafted design of a new DNN, auto-

[1] `https://dragon-tutorial.readthedocs.io/en/latest/index.html`

mated deep learning facilitates the creation of powerful DNNs. Interests are to give access to deep learning to less experienced people, to reduce the tedious tasks of managing many parameters to reach the optimal DNN, and finally, to go beyond what humans can design by creating non-intuitive DNNs that can ultimately prove to be more efficient.

Optimizing a DNN means automatically finding an optimal architecture for a given learning task: choosing the operations and the connections between those operations and the associated hyperparameters. The first task is known as Neural Architecture Search (Elsken et al., 2019), also named NAS, and the second, as Hyperparameters Optimization (HPO). Most works from the literature try to tackle only one of these two optimization problems. Many papers related to NAS (White et al., 2021; Loni et al., 2020; Wang et al., 2019; Sun et al., 2018; Zhong et al., 2020) focus on designing optimal architectures for computer vision tasks with stacked convolution and pooling layers. Because each DNN training is time-consuming, researchers tried to reduce the search space by adding many constraints preventing from finding irrelevant architectures. These strategies are relevant in the case of computer vision or NLP, where the models to be trained are huge and the high performance architectures are well identified. However, there is a gap in the literature regarding the use of NAS and HPO for problems where neural networks could be efficient, but the relevant models have not been clearly identified.

To fill this gap, we introduce DRAGON (for DiRected Acyclic Graphs OptimizatioN), a new optimization framework for DNNs based on the evolution of Directed Acyclic Graphs (DAGs). The encoding and the search operators are highly flexible and may be used with various deep learning and AutoML problems. We ran experiments on time series forecasting tasks and demonstrate on a large variety of datasets that DRAGON can find DNNs which outperform state-of-the-art handcrafted forecasters and AutoML frameworks. In summary, our contributions are as follows:

- The precise definition of a flexible search space based on DAGs, for the optimization of DNN architectures and hyperparameters. This search space may be used for various tasks, and is particularly useful when the performing architectures for a given problem are not clearly identified.
- The design of efficient neighbourhoods and variation operators for DAGs. With these operators, any metaheuristic designed for a mixed and variable-size search space can be applied. In this paper, we investigate the use of an asynchronous evolutionary algorithm.
- The validation of the algorithmic framework on a popular time series forecasting benchmark (Godahewa et al., 2021). We compare ourselves with 15 handcrafted statistical and machine learning models (Godahewa et al., 2021) as well as 6 AutoML frameworks on 27 datasets (Shchur et al., 2023). We show that DRAGON outperforms the 21 models from this baseline on 11 out of 27 datasets. The only competitive model is the AutoML framework AutoGluon (Shchur et al., 2023), which outperforms the baseline on 10 out of 27 datasets and was beaten by DRAGON on 14 out of 27 datasets.

The paper is organized as follows: we review section 2, the literature on deep learning models for time series forecasting and AutoML. Section 3 defines our search space.

Section 4 presents our neighbourhoods and variation operators within the evolutionary algorithm. Section 5 details our experimental results obtained on a popular time series forecasting benchmark. Finally, Section 6 gives a conclusion and introduces further research opportunities.

2. Related Work

2.1 Deep Learning for Time Series Forecasting

Time series forecasting has been studied for decades. The field has been dominated for a long time by statistical tools such as ARIMA, Exponential Smoothing (ES), or (S)ARIMAX, this last model allowing the use of exogenous variables. It now opens itself to deep learning models (Liu et al., 2021). These new models recently achieved great performances on many datasets. Three main parts compose typical DNNs: an input layer, several hidden layers and an output layer. In this paper we apply our framework to optimize the hidden layers for a given time series forecasting task (see Figure 5). In this part, we introduce usual DNN layers for time series forecasting, which can be used in our search space.

The first layer type from our search space is the fully-connected layer, or Multi-Layer Perceptron (MLP). The input vector is multiplied by a weight matrix. Most architectures use such layers as simple building blocks for dimension matching, input embedding or output modelling. The N-Beats model is a well-known example of a DNN based on fully-connected layers for time series forecasting (Oreshkin et al., 2019).

The second layer type (LeCun et al., 2015) is the convolution layer (CNN). Inspired by the human brain's visual cortex, it has mainly been popularised for computer vision. The convolution layer uses a discrete convolution operator between the input data and a small matrix called a filter. The extracted features are local and time-invariant if the considered data are time series. Many architectures designed for time series forecasting are based on convolution layers such as WaveNet (Oord et al., 2016) and Temporal Convolution Networks (Lea et al., 2017).

The third layer type is the recurrent layer (RNN), specifically designed for sequential data processing, therefore, particularly suitable for time series. These layers scan the sequential data and keep information from the sequence past in memory to predict its future. A popular model based on RNN layers is the Seq2Seq network (Cho et al., 2014). Two RNNs, an encoder and a decoder, are sequentially connected by a fixed-length vector. Various versions of the Seq2Seq model have been introduced in the literature, such as the DeepAR model (Salinas et al., 2020), which encompasses an RNN encoder in an autoregressive model. The major weakness of RNN layers is the modelling of long-term dynamics due to the vanishing gradient. Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) layers have been introduced (Hochreiter and Schmidhuber, 1997; Chung et al., 2014) to overcome this problem.

Finally, the layer type from our search space is the attention layer. The attention layer has been popularized within the deep learning community as part of Vaswani's transformer model (Vaswani et al., 2017). The attention layer is more generic than the convolution. It can model the dependencies of each element from the input sequence with all the others. In the vanilla transformer (Vaswani et al., 2017), the attention layer does not factor the relative distance between inputs in its modelling but rather the element's absolute position in the sequence. The Transformer-XL (Dai et al., 2019), a transformer variant created to tackle long-term dependencies tasks, introduces a self-attention version with relative positions. Cordonnier et al. (2019) used this new attention formulation to show that, under a specific configuration of parameters, the attention layers could be trained as convolution layers. Within our search space, we chose this last formulation of attention, with the relative positions.

The three first layers (i.e. MLP, CNN, RNN) were frequently mixed into DNN architectures. Sequential and parallel combinations of convolution, recurrent and fully connected layers often compose state-of-the-art DNN models for time series forecasting. Layer diversity enables the extraction of different and complementary features from input data to allow a better prediction. Some recent DNN models introduce transformers into hybrid DNNs. In Lim et al. (2021), the authors developed the Temporal Fusion Transformer, a hybrid model stacking transformer layers on top of an RNN layer. With this in mind, we built a flexible search space which generalizes hybrid DNN models including MLPs, CNNs, RNNs and transformers.

2.2 Search Spaces for Automated Deep Learning

Designing an efficient DNN for a given task requires choosing an architecture and tuning its many hyperparameters. It is a difficult, fastidious, and time-consuming optimization task. Moreover, it requires expertise and restricts the discovery of new DNNs to what humans can design. Research related to the automatic design and optimization of DNNs has therefore risen this last decade (Talbi, 2021). The first challenge in automatic deep learning (AutoDL), and more specifically neural architecture search (NAS), is search space design. Typical search spaces for Hyperparameters Optimization (HPO) are a product space of a mixture of continuous and categorical dimensions (e.g. learning rate, number of layers, batch size), while NAS focuses on optimizing the topology of the DNN (White et al., 2023). Encoding a DNN topology is a complex task because the encoding should not be too broad and allow too many architectures to keep the search efficient. On the contrary, if the encoding is too restrictive, we may miss promising solutions and novel architectures. This means before creating the search space we need to choose which DNNs or type of DNNs are relevant or not to the problem at hand. Once we have decided on this broad set of DNNs, we define the search space following a set of rules (Talbi, 2021):

- Completeness: all (or almost all) relevant DNNs from this broad set should be encoded in the search space.
- Connectedness: a path should always be possible between two encoded DNNs in the search space.
- Efficiency: the encoding should be easy to manipulate by the search operators (i.e. neighbourhoods, variation operators) of the search strategy.
- Constraint handling: the encoding should facilitate the handling of the various constraints to generate feasible DNNs.

A complete classification of encoding strategies for NAS is presented in Talbi (2021) and reproduced in Figure 1. We can discriminate between direct and indirect encodings.

With direct strategies, the DNNs are completely defined by the encoding, while indirect strategies need a decoder to find the architecture back. Amongst direct strategies, one can discriminate between two categories: flat and hierarchical encodings. In flat encodings, all layers are individually encoded (Loni et al., 2020; Sun et al., 2018; Wang et al., 2018, 2019). The global architecture can be a single chain, with each layer having a single input and a single output, which is called chain structured (Assunção et al., 2019), but more complex patterns such as multiple outputs, skip connections, have been introduced in the extended flat DNNs encoding (Chen et al., 2021). For hierarchical encodings, they are bundled in blocks (Pham et al., 2018; Shu et al., 2019; Liu et al., 2017; Zhang et al., 2019). If the optimization is made on the sequencing of the blocks, with an already chosen content, this is referred to as inner-level fixed (Camero et al., 2021; White et al., 2021). If the optimization is made on the blocks' content with a fixed sequencing, it is called outer level fixed. A joint optimization with no level fixed is also an option (Liu et al., 2019). Regarding the indirect strategies, one popular encoding is the one-shot architecture (Bender et al., 2018; Brock et al., 2017). One single large network resuming all candidates from the search space is trained. Then the architectures are found by pruning some branches. Only the best promising architectures are retrained from scratch.

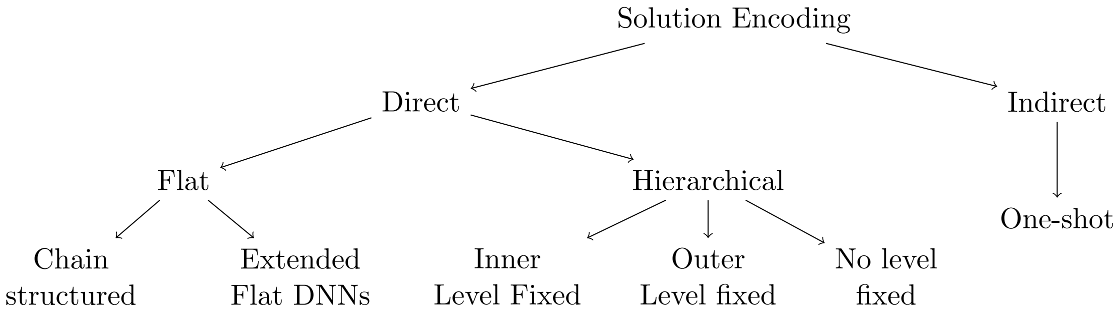


Figure 1: Classification of encoding strategies for NAS (Talbi, 2021).

Our search space can be categorized as a direct and extended flat encoding. It is based on the representation of DNNs by DAGs. This representation is very popular among the NAS community and is used by cell-based search spaces such as NAS-Bench-101 inspired by the ResNet architecture (Ying et al., 2019), as well as one-shot representation such as the DARTS framework (for Differentiable Architecture Search) proposed by Liu et al. (2018b). In cell-based search spaces, DNNs are represented by repeated cells encoded as DAGs, where each node is an operation belonging to a well-defined list, typically: convolution of size 1, 3, or 5, pooling of size 3, skip connection, or zeroed operation for an image classification task for example. The graphs are then represented either as vectors using path encoding, or as adjacency matrices. In the case of path encoding, different search algorithms can be used, such as Bayesian optimization (White et al., 2021), reinforcement learning (Zoph et al., 2018), particle swarm optimization (Wang et al., 2019), or evolutionary algorithms (Xie and Yuille, 2017), for which classical mutation and crossover operators are usually used and consist in modifying the elements of the path. Adjacency matrices, on the other hand, are more complex objects to optimize. The matrix itself represents the connections within the graph and is usually accompanied by a list representing the nodes content. In the literature, these matrices have been optimized directly with random search algorithms (Irwin-Harris et al., 2019) or indirectly with neural predictors based on auto-encoders (see for example Zhang et al. (2019) or Chatzianastasis et al. (2021)). In the case of one-shot representations, an initial large graph containing all the considered DNN is pruned with a certain search algorithm to only keep the best possible subgraph (and thus the best possible DNN). Various search algorithms can be used to simplify this meta-graph (Bender et al., 2018) like evolutionary algorithm (Guo et al., 2020). One of the most widely used techniques is DARTS (Liu et al., 2018b), where each edge is associated with a candidate operation, assigned to a probability of being retained in the final subgraph, optimized by gradient descent. The candidate operations can be quite commons: convolution or pooling layers for instance, such as for cell-based search spaces, but Chen et al. (2021) proposes to use DARTS with other types of operations, such as inter-variable attention, for multivariate time series prediction. While cell-based and one-shot architecture search spaces have proven to be efficient for tasks like image classification or language processing, White et al. (2023) pointed out that current NAS search spaces are not very expressive and prevents finding highly novel architectures. This problem is amplified when dealing with tasks for which no known architectures have yet been found.

Compared to these search spaces, the one we define in this paper is more flexible. We address the optimization of both the architecture and the hyperparameters. We do not fix a list of possible operations with fixed hyperparameters, as is done in these works, but leave the user free to use any operation coded as *PyTorch nn.Module* and to optimize any chosen parameters. Furthermore, we do not fix the generic form of our graph as a maximum number of incoming or outgoing edges and we allow to expand or reduce the graphs. DRAGON is capable of generating innovative, original, yet well-performing DNNs. This flexibility may hinder the framework's ability to find good DNNs compared to the NAS state-of-the-art for well-known tasks such as image classification or language processing. However, in cases where DNNs have not been extensively studied and well-performing architectures have not yet been found, such as time series forecasting, DRAGON may be an efficient DNN designer. Finally, we encode our DAGs using their adjacency matrices and provide evolutionary operators to directly modify this representation. To our knowledge, neither such a large search space nor such operators have been used in the literature.

2.3 AutoML for Time Series Forecasting

The automated design of DNNs called Automated Deep Learning (AutoDL), belongs to a larger field (Hutter et al., 2019) called Automated Machine Learning (AutoML). AutoML aims to automatically design well-performing machine learning pipelines, for a given task. Works on model optimization for time series forecasting mainly focused on AutoML rather than AutoDL (Alsharef et al., 2022). The optimization can be performed at several levels: input features selection, extraction and engineering, model selection and hyperparameters tuning. Initial research works used to focus on one of these subproblems, while more recent works offer complete optimization pipelines.

The first subproblems, input features selection, extraction and engineering, are specific to our learning task: time series forecasting. This tedious task can significantly improve the prediction scores by giving the model relevant information about the data. Methods to select the features are among computing the importance of each feature on the results or using statistical tools on the signals to extract relevant information. Next, the model selection aims at choosing among a set of diverse machine learning models the best-performing one on a given task. Often, the models are trained separately, and the best model is chosen. In general, the selected model has many hyperparameters, such as the number of hidden layers, activation function or learning rate. Their optimization usually allows for improving the performance of the model.

Nowadays, many research works implement complete optimization pipelines combining those subproblems for time series forecasting. The Time Series Pipeline Optimization framework (Dahl, 2020), is based on an evolutionary algorithm to automatically find the right features thanks to input signal analysis, then the model and its related hyperparameters. AutoAI-TS (Shah et al., 2021) is also a complete optimization pipeline, with model selection performed among a wide assortment of models: statistical models, machine learning, deep learning models and hybrids models. Closer to our work, the framework AutoPytorch-TS (Deng et al., 2022) is specific to deep learning models optimization for time series forecasting. The framework uses Bayesian optimization with multi-fidelity optimization. Finally, a recent work from Amazon (Shchur et al., 2023) introduces a time series version to their AutoML framework, AutoGluon, leveraging ensembles of statistical and machine learning forecasters.

Except for AutoPytorch-TS, cited works covering the entire optimization pipeline for time series do not deepen model optimization and only perform model selection and hyperparameters optimization. However, time series data becomes more complex, and there is a growing need for more sophisticated and data-specific DNNs. Our framework DRAGON, presented in this paper, only tackles the model selection and hyperparameters optimization parts of the pipeline. We made this choice to show the effectiveness of our framework for designing better DNNs. If we had implemented feature selection, it would have been harder to determine whether the superiority of our results came from the input features pool or the model itself.

3. Search Space Definition

The development of our optimization framework DRAGON requires the definition of a search space, an objective function and a search algorithm. In this section, we formulate the handled optimization problem an then we detail DRAGON's search space and its characteristics.

3.1 Optimization Problem Formulation

Our optimization problem consists in finding the best possible DNN for a given time series forecasting problem. To do so, we introduce an ensemble Ω representing our search space, which contains all considered DNNs. We then consider our time series dataset $\mathcal{D}$. For any subset $\mathcal{D}_0 = (X_0, Y_0) \in \mathcal{P}(\mathcal{D})$, we define the forecast error ℓ as:

$$\begin{aligned}\ell \colon \Omega \times \mathcal{P}(\mathcal{D}) &\to \mathbb{R} \\ f \times \mathcal{D}_0 &\mapsto \ell\big(f(\mathcal{D}_0)\big) = \ell\big(Y_0, f(X_0)\big)\,.\end{aligned}$$

The explicit formula for ℓ will be given later in the paper. Each element f from Ω is a DNN defined as an operator parameterized by three parameters. First, its architecture $\alpha \in \mathcal{A}$. The search space of all considered architectures is called $\mathcal{A}$ and will be detailed in Subsection 3.2. Given the DNN architecture α, the DNN is then parameterized by its hyperparameters $\lambda \in \Lambda(\alpha)$, with $\Lambda(\alpha)$ the search space of the hyperparameters induced by the architecture α and defined Subsection 3.3. Finally, α and λ generate an ensemble of possible weights $\Theta(\alpha, \lambda)$, from which the DNN optimal weights θ are found by gradient descent when training the model. The architecture α and the hyperparameters λ are optimized by our framework DRAGON.

We consider the multivariate time series forecasting task. Our dataset $\mathcal{D} = (X, Y)$ is composed of a target variable $Y = \{\mathbf{y}_t\}_{t=1}^T$, with $\mathbf{y}_t \in \mathbb{R}^N$ the target value at the time step t, and a set of explanatory variables (features) $X = \{\mathbf{x}_t\}_{t=1}^T$, with $\mathbf{x}_t \in \mathbb{R}^{F_1 \times F_2}$. The size of the target Y at each time step is N and F_1, F_2 are the shapes of the input variable X at each time step. We choose to represent $\mathbf{x}_t$ by a matrix to extend our framework's scope, but it can equally be defined as a vector by taking $F_2 = 1$. DRAGON can be applied to univariate signals by taking $N = 1$. We partition our time indexes into three groups of successive time steps and split accordingly $\mathcal{D}$ into three datasets: $\mathcal{D}_{train}$, $\mathcal{D}_{valid}$ and $\mathcal{D}_{test}$.

After choosing an architecture α and a set of hyperparamaters λ, we build the DNN $f^{\alpha,\lambda}$ and use $\mathcal{D}_{train}$ to train $f^{\alpha,\lambda}$ and optimize its weights θ by stochastic gradient descent:

$$\hat{\theta} \in \underset{\theta \in \Theta(\alpha,\lambda)}{\arg\min}\big(\ell(f_\theta^{\alpha,\lambda}, \mathcal{D}_{train})\big).$$

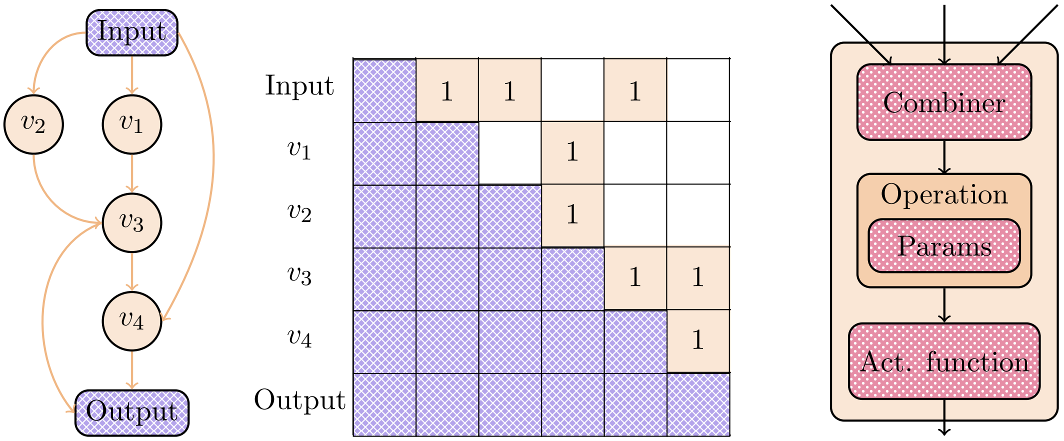
The forecast error of the DNN parameterized by $\hat{\theta}$ on $\mathcal{D}_{valid}$ is used to assess the performance of the selected α and λ. The best architecture and hyperparameters are optimized by solving:

$$(\hat{\alpha}, \hat{\lambda}) \in \underset{\alpha \in \mathcal{A}}{\arg\min}\Big(\underset{\lambda \in \Lambda(\alpha)}{\arg\min}\big(\ell(f_{\hat{\theta}}^{\alpha,\lambda}, \mathcal{D}_{valid})\big)\Big).$$

The function $(\alpha, \lambda) \mapsto \ell(f_{\hat{\theta}}^{\alpha,\lambda}, \mathcal{D}_{valid})$ corresponds to the objective function of DRAGON. We finally will evaluate the performance of DRAGON by computing the forecast error on $\mathcal{D}_{test}$ using the DNN with the best architecture, hyperparameters and weights:

$$\ell(f_{\hat{\theta}}^{\hat{\alpha},\hat{\lambda}}, \mathcal{D}_{test}).$$

In practice, the second equation optimizing α and λ can be solved separately or jointly. If we fix λ for each α, the optimization is made only on the architecture and is referred to as Neural Architecture Search (NAS). If α is fixed, then the optimization is only made on the model hyperparameters and is referred to as Hyperparameters Optimization (HPO). DRAGON allows to fix α or λ during parts of the optimization to perform a hierarchical optimization: ordering optimisation sequences during which only the architecture is optimised, and others during which only the hyperparameters are optimised. In the following, we will describe our search space $\Omega = (\mathcal{A} \times \{\Lambda(\alpha), \alpha \in \mathcal{A}\})$.



Input	▨	1	1		1	
v_1	▨	▨		1		
v_2	▨	▨	▨	1		
v_3	▨	▨	▨	▨	1	1
v_4	▨	▨	▨	▨	▨	1
Output	▨	▨	▨	▨	▨	▨

(a) Architecture (b) Adjacency matrix representation (c) Inside node v_i

Figure 2: DNN encoding as a Directed Acyclic Graph (DAG). The elements in blue (crosshatch) are fixed by the framework, the architecture elements from α are displayed in beige and the hyperparameters λ are in pink (dots).

3.2 Architecture Search Space

First, we define our architecture search space $\mathcal{A}$. We propose to model a DNN by a Directed Acyclic Graph (DAG) with a single input and output (Fiore and Devesas Campos, 2013). A DAG $\Gamma = (\mathcal{V}, \mathcal{E})$ is defined by its nodes (or vertices) set $\mathcal{V} = \{v_1, ..., v_n\}$ and its edges set $\mathcal{E} \subseteq \{(v_i, v_j)|v_i, v_j \in \mathcal{V}\}$. Each node v represents a DNN layer as defined in Subsection 2.1, such as a convolution, a recurrence, or a matrix product. To eliminate isolated nodes, we impose each node to be connected by a path to the input and the output. The graph acyclicity implies a partial ordering of the nodes. If a path exists from the node v_a to a node v_b, then we can define a relation order between them: $v_a < v_b$. Acyclicity prevents the existence of a path from v_b to v_a. However, this relation order is not total. When dealing with symmetric graphs where all nodes are not connected, several nodes' ordering may be valid for the same graph. For example in Figure 2a, the orderings $v_1 > v_2$ and $v_2 > v_1$ are both valid.

Hence, a DAG Γ is represented by a sorted list $\mathcal{L}$, such that $|\mathcal{L}| = m$, containing the graph nodes, and its adjacency matrix $M \in \{0, 1\}^{m \times m}$ (Zhang et al., 2019). The matrix M is built such that: $M(i, j) = 1 \Leftrightarrow (v_i, v_j) \in \mathcal{E}$. Because of the graph's acyclicity, the matrix is upper triangular with its diagonal filled with zeros. The input node has no incoming connection, and the output node has no outcoming connection, meaning $\sum_{i=1}^{m} M_{i,1} = 0$ and $\sum_{j=1}^{m} M_{m,j} = 0$. Besides, the input is necessarily connected to the first node and the last node to the output for any graph, enforcing: $M_{1,2} = 1$ and $M_{m-1,m} = 1$. As isolated nodes do not exist in the graph, we need at least a non-zero value on every row and column, except for the first column and last row. We can express this property as: $\forall i < m : \sum_{j=i+1}^{m} M_{i,j} > 0$ and $\forall j > 1 : \sum_{i=j+1}^{m} M_{i,j} > 0$. Finally, the ordering of the partial nodes does not allow a bijective encoding: several matrices M may encode the same DAG.

To summarize, we have $\mathscr{A} = \{\Gamma = (\mathscr{V}, \mathscr{E}) = (\mathscr{L}, M)\}$. The graphs Γ are parameterized by their size m which is not equal for all graphs. As we will see in Section 4.1 the DNNs size may vary during the optimization.

3.3 Hyperparameters Search Space

For any fixed architecture $\alpha \in \mathscr{A}$, let's define our hyperparameters search space induced by $\alpha : \Lambda(\alpha)$. As mentioned above, the DAG nodes represent the DNN hidden layers. A set of hyperparameters λ, also called a graph node, is composed of a combiner, a layer operation and an activation function (see Figure 2c). Each layer operation is associated with a specific set of parameters, like output or hidden dimensions, convolution kernel size or dropout rate. We provide in Appendix A a table with all available operations and their associated parameters. The hyperparameters search space $\Lambda(\alpha)$ is made of sets λ composed with a combiner, the layer's parameters and the activation function.

First, we need a combiner as each node can receive an arbitrary number of input connections. The parents' latent representations should be combined before being fed to the operation. Taking inspiration from the Google Brain Team Evolved Transformer (So et al., 2019), we propose three types of combiners: element-wise addition, element-wise multiplication and concatenation. The input vectors may have different channel numbers and the combiner needs to level them. This issue is rarely mentioned in the literature, where authors prefer to keep a fixed channel number (Liu et al., 2018b). In the general case, for element-wise combiners, the combiner output channel matches the maximum channel number of latent representation. We apply zero-padding on the smaller inputs. For the concatenation combiner, we consider the sum of the channel number of each input. Some operations, for instance, the pooling and the convolution operators, have kernels. Their calculation requires that the number of channels of the input vector is larger than this kernel. In these cases, we also perform zero-padding after the combiner to ensure that we have the minimum number of channels required. We use the node order to create our DNN. We build the nodes one at a time, following their order in $\mathscr{L}$. The first node is created using the data input shape as argument. After its creation we compute and gather its output shape. Then, for each following node, we compute the layer operation input shape according to the output shapes of the connected nodes and the combiner. After building the operation we compute its output shape for the next layers. Finally, as depicted in Figure 2c, each node ends with an activation function. The hyperparameters optimized for each node can be found Annex A.

To summarize, we define every node as the sequence of combiner → layer → activation function. In our search space $\Lambda(\alpha)$, the nodes are encoded by Python objects having as attributes the combiner name, the layer corresponding to the operation set with the hyperparameters encoded as a PyTorch Module followed by the activation function. The set $\mathscr{L}$ is a variable-length list containing each node.

4. Search Algorithm

The search space from DRAGON $\Omega = (\mathscr{A} \times \{\Lambda(\alpha), \alpha \in \mathscr{A}\})$ defined in the previous section is a mixed and variable space: it may contain integers, float, and categorical values, and the dimension of its elements, the DNNs, is not fixed. We need to design a search algorithm able to efficiently navigate through this search space. While several metaheuristics can solve mixed and variable-size optimization problems (Talbi, 2023), we chose to start with an evolutionary algorithm. This metaheuristic was the most intuitive for us to manipulate Directed Acyclic Graphs. It has been used to optimize graphs in other fields, for example on logic circuits (Aguirre and Coello Coello, 2003). In Section 5 we compare our model to other simple metaheuristics: the Random Search and the Simulated Annealing, but the design of more complex metaheuristics using our search space and their comparison with the evolutionary algorithm are left to future work.

4.1 Evolutionary Algorithm Design

Evolutionary algorithms represent popular metaheuristics which are well adapted to solve mixed and variable-space optimization algorithms (Talbi, 2023). They have been widely used for the automatic design of DNNs (Li et al., 2022). The idea is to evolve a randomly generated population of DAGs to converge towards an optimal DNN. An optimal solution for a time series forecasting task is defined as a DNN minimizing a forecasting error. As training a DNN is expensive in time and computational resources, we implemented an asynchronous version, also called steady-state, of the evolutionary algorithm. This version is more efficient on High-Performance Computing (HPC) systems as detailed by Liu et al. (2018a). At the beginning of the algorithm, a set of K random DNNs is generated. Each solution is train on $\mathscr{D}_{train}$ and evaluate on $\mathscr{D}_{valid}$ to create a population of size K. Then, for a certain number of iterations or a fixed time budget B, once a processus is free, a selection operator selects two solutions from the population. Those solutions are modified using crossover and mutation operators to create two offsprings. Those are trained and evaluated by the free process. Then, for each offspring, if its loss ℓ is less than the worst loss from the population, the offspring replaces the worst individual. Using an asynchronous version instead of the classical one avoids waiting for a whole generation to be evaluated and saves some time. The complete flowchart is shown in Figure 3.

DRAGON's search space defined Section 3 is not directly efficient with common mutation and crossover operators. Therefore, we had to define evolution operators specific for our search space. Those operators can be used with various metaheuristics: a mutation operator for example can be used as a neighborhood operator for a local search. We split the operators into two categories: hyperparameters specific operators and architecture operators. The idea is to allow a sequential or joint optimization of the hyperparameters and the architecture. All the candidate operations which can be used in the the graphs nodes do not share the same hyperparameters. Thus, drawing a new layer means modifying all its parameters and one can lose the optimization made on the hyperparameters of the previous operation. Using sequential optimization, the algorithm can first find well-performing architectures and operations during the architecture search and then fine-tune the found DNNs during the hyperparameters search.

4.2 Architecture Evolution

In this section, we introduce the architecture-specific search operators from DRAGON. By architecture, we mean the search space $\mathscr{A}$ defined above: the nodes' operation and the edges between them.

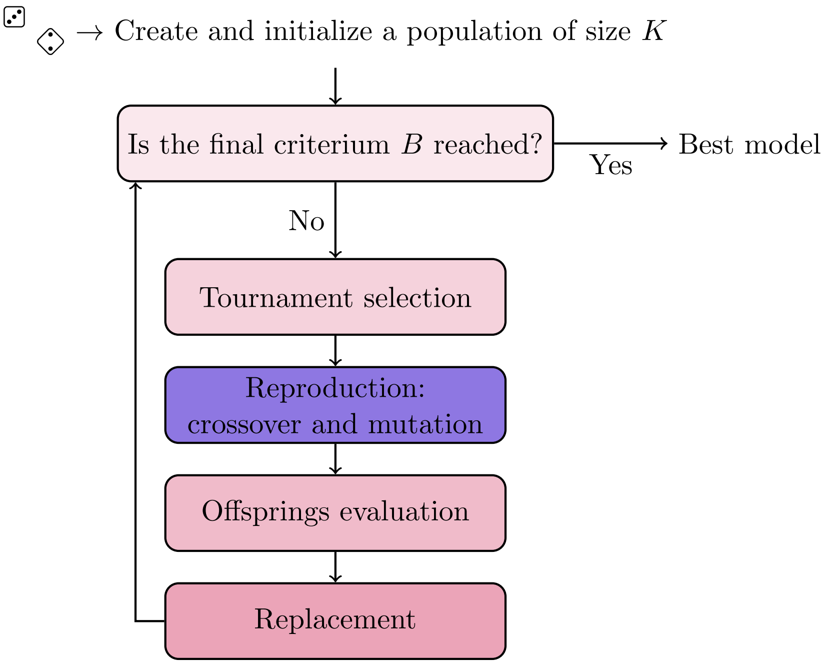

→ Create and initialize a population of size K
Is the final criterium B reached?
Yes
Best model
No
Tournament selection
Reproduction: crossover and mutation
Offsprings evaluation
Replacement

Figure 3: Evolutionary algorithm flowchart.

Mutation. The mutation operators are simple modifications inspired by the Graph Edit Distance (Abu-Aisheh et al., 2015): insertion, deletion and substitution of both nodes and edges. Given a graph $\Gamma = (\mathcal{L}, M)$, the mutation operator will draw the set $\mathcal{L}' \subseteq \mathcal{L}$ and apply a transformation to each node of $\mathcal{L}'$. Let's have $v_i \in \mathcal{L}'$ the node that will be transformed:

- *Node insertion:* we draw a new node with its combiner, operation and activation function. We insert the new node in our graph at the position $i+1$. We draw its incoming and outgoing edges by verifying that we do not generate an isolated node.

- *Node deletion:* we delete the node v_i. In the case where it generates other isolated nodes, we draw new edges.

- *Parents modification:* we modify the incoming edges for v_i and make sure we always have at least one.

- *Children modification:* we modify the outgoing edges for v_i and make sure we always have at least one.

- *Node modification:* we draw the new content of v_i, the new combiner, the operation and/or the activation function.

Thanks to these mutation operators, we make our search space connected, as explained in Section 2. In fact, by successively using these operations, we can move from any graph to any other, since the Graph Edit Distance can be used with any pair of graphs.

Crossover. The second architecture-specific operator we implemented is the crossover. The idea is to inherit patterns from two parents to create two offsprings. The original crossover is applied to two vectors. It exchanges two parts of these vectors. In our case, the individuals are graphs. Let's say we have two parents Γ_1 and Γ_2. The first step is to randomly select one subgraph from each parent, $\gamma_1 \subset \Gamma_1$ and $\gamma_2 \subset \Gamma_2$ to exchange (see figure 4a). Next the two offspring Γ_1' and Γ_2' are generated from Γ_1 and Γ_2 by removing γ_1 and γ_2, as shown in figure 4b. Next, we need to define the position at which each of the subgraphs will be inserted into the host graph. The idea is to preserve the overall structure of the graph. In other words, if the subgraph was at the beginning of the parent graph, it should also be at the beginning of the child graph, and vice versa. We denote here, for a node $v \in \Gamma$, $p(v, \Gamma)$ its position in the graph Γ, and $P(\Gamma) = \{p(v, \Gamma), v \in \Gamma\}$ the set of all nodes positions in Γ. We compute the future positions of each node $v \in \gamma_1$ in Γ_2' sequentially, starting with the first node, $v_1 \in \arg\min_{v \in \gamma_1} p(v, \gamma_1)$. The position $p(v_1, \Gamma_2')$ of v_1 in the graph Γ_2' can be computed as:

$$p(v_1, \Gamma_2') \in \underset{p \in P(\Gamma_2')}{\arg\min}(|p - p(v_1, \Gamma_1)|)$$

The positions from the following nodes $\{v_2, ..., v_g\} \in \gamma_1$ are computed to respect the structure of Γ_1 and γ_1:

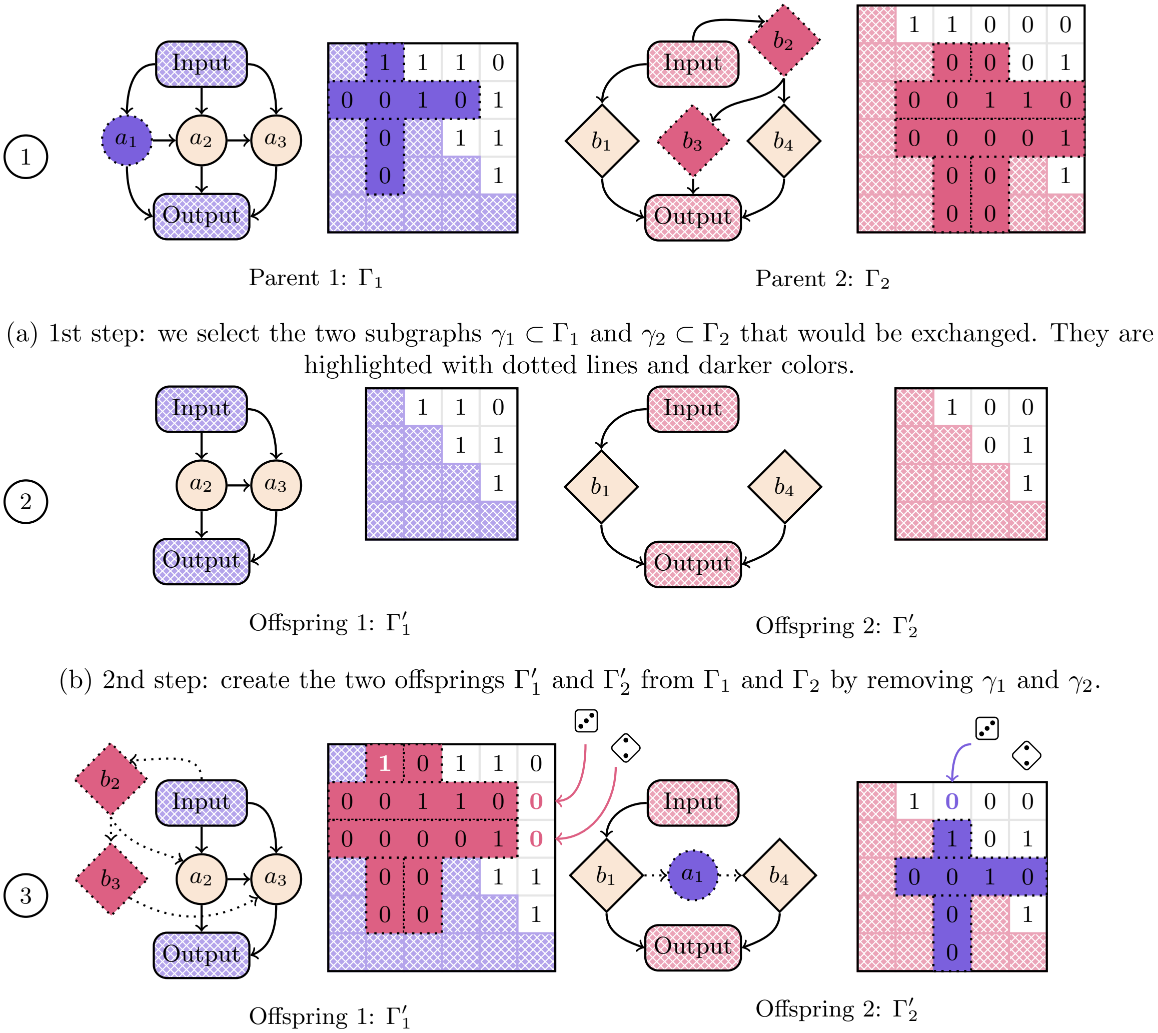
$$p(v_i, \Gamma_2') = \min\left(p(v_i, \Gamma_1) - p(v_{i-1}, \Gamma_1) + p(v_{i-1}, \Gamma_2'), |\Gamma_2'| + |\gamma_1|\right)$$

Finally, as shown Figure 2a, the rows and columns corresponding to the nodes from γ_1 and γ_2 are inserted in the adjacency matrices of Γ_2' and Γ_1' at the previously computed positions. If the process has generated orphan nodes, we randomly generate the necessary connections.

4.3 Hyperparameters Evolution

One of the architecture mutations consists in disturbing the node content. In this case, the node content is modified, including the operation. A new set of hyperparameters is then drawn. To refine this search, we defined specific mutations for the search space $\Lambda(\alpha)$. In the hyperparameters case, edges and nodes number are not affected. As for architecture-specific mutation, the operator will draw the set $\mathscr{L}' \subseteq \mathscr{L}$ and apply a transformation on each node of $\mathscr{L}'$. For each node v_i from $\mathscr{L}'$, we draw h_i hyperparameters, which will be modified by a neighbouring value. The hyperparameters in our search space belong to three categories:

- **Categorical values:** the new value is randomly drawn among the set of possibilities deprived of the actual value. For instance, the activation functions, combiners, and recurrence types (LSTM/GRU) belong to this type of categorical variable.
- **Integers:** we select the neighbours inside a discrete interval around the actual value. For instance, it has been applied to convolution kernel size and output dimension.
- **Float:** we select the neighbours inside a continuous interval around the actual value. Such a neighbourhood has been defined for instance to the dropout rate.



	1	1	1	0
0	0	1	0	1
	0		1	1
	0			1

	1	1	0	0	0
		0	0	0	1
	0	0	1	1	0
	0	0	0	0	1
		0	0		1
		0	0		

Parent 1: Γ_1

Parent 2: Γ_2

(a) 1st step: we select the two subgraphs $\gamma_1 \subset \Gamma_1$ and $\gamma_2 \subset \Gamma_2$ that would be exchanged. They are highlighted with dotted lines and darker colors.

	1	1	0
		1	1
			1

	1	0	0
		0	1
			1

Offspring 1: Γ'_1

Offspring 2: Γ'_2

(b) 2nd step: create the two offsprings Γ'_1 and Γ'_2 from Γ_1 and Γ_2 by removing γ_1 and γ_2.

	1	0	1	1	0
0	0	1	1	0	0
0	0	0	0	1	0
	0	0		1	1
	0	0			1

	1	0	0	0
		1	0	1
	0	0	1	0
		0		1
		0		

Offspring 1: Γ'_1

Offspring 2: Γ'_2

(c) 3rd step: we insert the nodes from γ_1 in Γ'_2 and the nodes from γ_2 in Γ'_1 and reconstruct the edges.

Figure 4: Crossover operator illustration.

5. Experimental Study

In this section, we describe how we evaluated DRAGON on a time series forecasting task. In the first three sections (5.1, 5.2 and 5.3), we define our experiments: the time series dataset we used, the models and AutoML frameworks we compared to DRAGON, and the meta-architecture we defined specifically for time series. Then, in the next three sections (5.4, 5.5 and 5.6) we present and analyze the results. Finally, the last two sections (5.7 and 5.8) discuss the limitations of the work and give some hints for further work.

5.1 Baseline

We compared our framework to two baselines. The first consists of 15 handcrafted models (Godahewa et al., 2021), the second is more recent and compares 6 AutoML frameworks specifically designed for time series forecasting (Shchur et al., 2023).

Handcrafted models. These are statistical, machine learning and deep learning models that were built and optimised by hand. We first have 5 traditional univariate forecasting models: Simple Exponential Smoothing (SES), Exponential Smoothing (ETS), Theta, Trigonometric Box-Cox ARMA Trend Seasonal (TBATS), Dynamic Harmonic Regression ARIMA (DHR-ARIMA) and 8 global forecasting models: Pooled Regression (PR), CatBoost, Prophet, Feed-Forward Neural Network (FFNN), N-BEATS, WaveNet, Transformer and DeepAR. The last 5 models are Deep Neural Networks. Refer to the original paper (Godahewa et al., 2021) and the Monash Time Series Forecasting Repository website[2] for more information about the models and their implementation. Finally, Shchur et al. (2023) also provides the univariate forecasting model SeasonalNaive and the global deep learning model Temporal Fusion Transformer (TFT).

AutoML frameworks. Shchur et al. (2023) compares 6 AutoML frameworks specifically designed for time series forecasting. They first used 4 AutoML frameworks that are based on automated tuning of statistical models: AutoARIMA, AutoETS, AutoTheta, and StatsEnsemble. The first three automatically tune the hyperparameters of the ARIMA, ETS and Theta models for each time series individually. The optimization of the parameters is based on an information criterion. The last one, StatEnsemble, takes the median of the predictions of three statistical models. Then, they included the AutoDL framework AutoPyTorch-Forecasting, which optimizes the architecture and hyperparameters of DNNs using a combination of Bayesian and multi-fidelity optimization and then uses the model ensemble. Finally, AutoGluon-TS, the AutoML framework proposed by Shchur et al. (2023), relies on the ensemble techniques of local models such as ARIMA, Theta, ETS, and SeasonalNaive, as well as global models such as DeepAR, PatchTST, and Temporal Fusion Transformer. While it is interesting to compare ourselves with these state-of-the-art AutoML techniques, it is worth remembering that our framework does not yet provide an ensembling technique, and the scores obtained during optimization are based on the predictions of a single DNN.

[2] `https://forecastingdata.org/`

5.2 Experimental Protocol

We evaluated DRAGON on the established benchmark of Monash Time Series Forecasting Repository (Godahewa et al., 2021). This archive contains a benchmark of more than 40 datasets, from which we selected the 27 that Shchur et al. (2023) used for their experiments. The time series are of different kinds and have variable distributions. More information on each dataset from the archive is available Section B. This task diversity allows to test DRAGON generalization and robustness abilities.

For these experiments, we configured our algorithm to have a population of $K = 100$ individuals and we set the total budget to $B = 8$ hours. We investigated a joint optimization of the architecture α and the hyperparameters λ. We ran our experiments on 5 cluster nodes, each equipped with 4 Tesla V100 SXM2 32GB GPUs, using PyTorch 1.11.0 and Cuda 10.2.

We took the data, the data generation functions, the training parameters (batch size, number of epochs, learning rate), the training and prediction functions from the Monash Time Series Forecasting Repository, and we only changed the models themselves. We also kept for each time series the forecast horizon and the lag used in the repository. We believe our comparison is fair to the handcrafted and automatically designed models. Finally, to evaluate the models' performance, we used the same metric and metric implementation as in the repository. This metric represents the forecast error ℓ, and is the Mean Absolute Scaled Error (MASE), an absolute mean error divided by the average difference between two consecutive time steps (Hyndman and Koehler, 2006). Given a time series $Y = (\mathbf{y}_1, ..., \mathbf{y}_n)$ and the predictions $\hat{Y} = (\hat{\mathbf{y}}_1, ..., \hat{\mathbf{y}}_n)$, the MASE can be defined as:

$$\text{MASE}(Y, \hat{Y}) = \tfrac{n-1}{n} \times \tfrac{\sum_{t=1}^{n} |\mathbf{y}_t - \hat{\mathbf{y}}_t|}{\sum_{t=2}^{n} |\mathbf{y}_t - \mathbf{y}_{t-1}|}.$$

In our case, for $f \in \Omega$, $\mathcal{D}_0 = (X_0, Y_0) \subseteq \mathcal{D}$, we have $\ell\big(Y_0, f(X_0)\big) = \text{MASE}\big(Y_0, f(X_0)\big)$.

5.3 Search Space

The generic search space defined Section 3 introduces a brick, the Directed Acyclic Graph, which cannot directly be our search space. We used it to define a meta-architecture as represented Figure 5, which can directly replace the repository's models. The meta-architecture begins with the DAG Γ, which may be composed with various one-dimensional candidate operations (e.g. 1D convolution, LSTM, MLP). They can be found with their associated hyperparameters Table A. The DAG Γ is followed by a Multi-layer Perceptron (MLP) used to retrieve the time series output dimension, as the number of channels may vary within Γ. This search space is designed specifically for time series forecasting, but it could be modified for other tasks. For example if we want to use it for image classification, we would need a first graph with two-dimensional candidate operations, followed by a flatten layer, followed by a second graph with one-dimensional candidate operations and a final MLP layer.

5.4 Results

We report a summary of the results in Table 1. According to this summary, DRAGON outperforms all algorithms on 11 out of 27 datasets (41%). The second best algorithm, AutoGluon, was the only algorithm able to beat DRAGON on more than a third of the datasets. The direct competitor of DRAGON, namely AutoPytorch which is another Au-

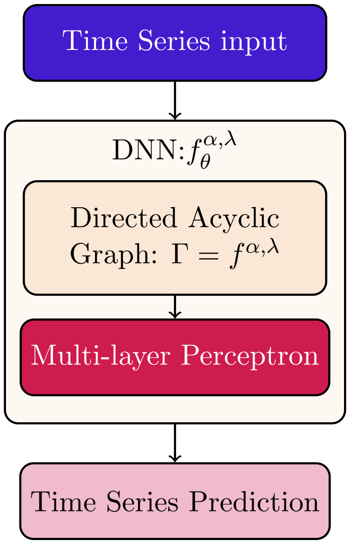
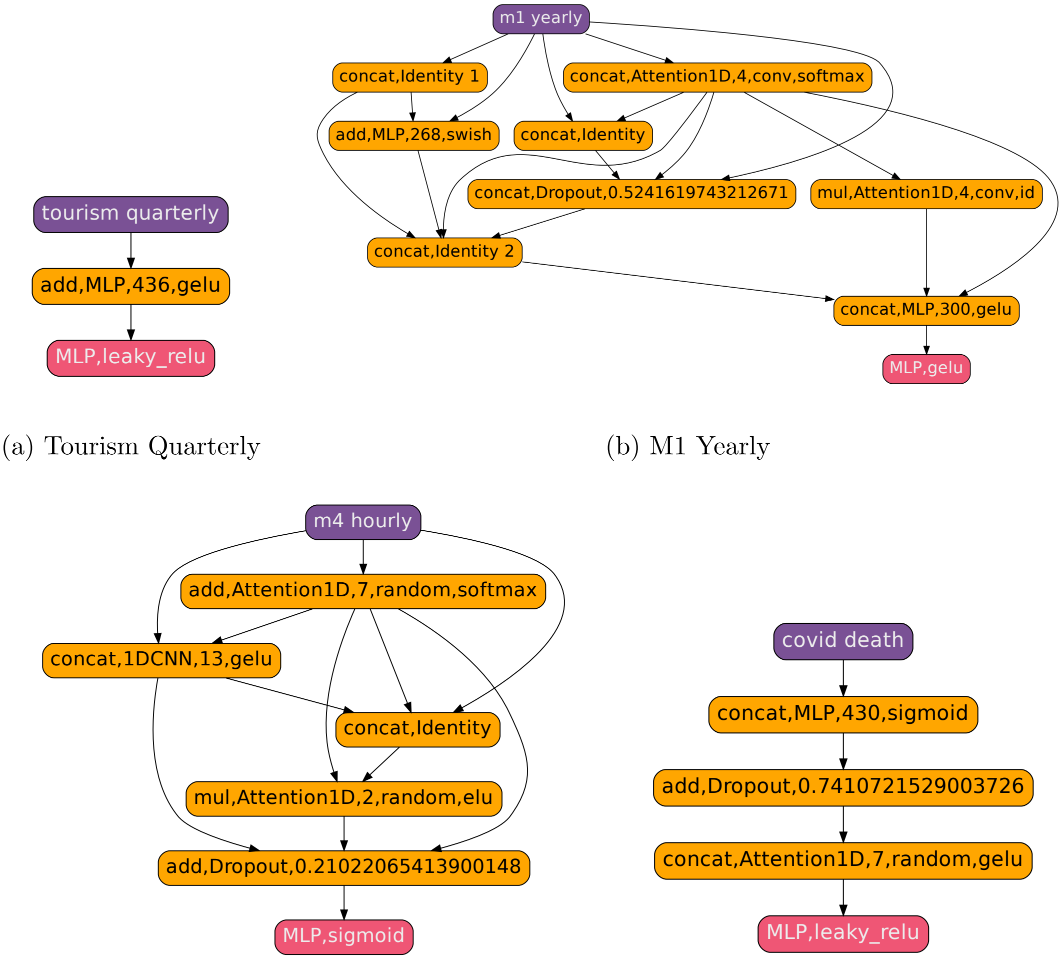

Time Series input
DNN:$f_\theta^{\alpha,\lambda}$
Directed Acyclic Graph: $\Gamma = f^{\alpha,\lambda}$
Multi-layer Perceptron
Time Series Prediction

Figure 5: Meta-architecture for Monash time series datasets.

Algorithm(s)	Wins	Losses	Champion	Failures
SES	1	25	0	1
Theta	6	20	0	1
TBATS	6	20	0	1
ETS	8	18	1	1
(DHR-)ARIMA	4	21	0	2
PR	1	25	0	1
CatBoost	1	25	0	1
FFNN	0	26	0	1
N-BEATS	0	26	0	1
WaveNet	2	23	0	2
Transformer	0	26	0	1
DeepAR	1	25	1	1
TFT	4	23	0	0
SeasonalNaive	1	26	0	0
Prophet	2	24	1	1
AutoPytorch	7	20	0	0
AutoARIMA	4	20	0	3
AutoETS	8	19	0	0
AutoTheta	9	16	0	2
StatEnsemble	9	15	3	3
AutoGluon	13	14	10	0
DRAGON	-	-	11	0

Table 1: Performance comparison of the baseline algorithms with DRAGON (based on the MASE metric) on 27 datasets. Wins corresponds to the number of datasets where the method produced a smaller loss than DRAGON, Losses corresponds to the number of datasets where the method produced a larger loss than DRAGON, Champion corresponds to the number of datasets where the method produced the smallest loss, and Failures corresponds to the number of datasets where the method failed.

toDL framework, was only able to beat it on 7 datasets out of 27 (26%). More detailed results can be found in Table 2, and visual representations of the DNNs found for some time series can be found in Figure 6. The content of the Γ graph is shown in yellow, while the last MLP layer is shown in pink.



(a) Tourism Quarterly

(b) M1 Yearly

(c) M4 Hourly

(d) Covid Death

Figure 6: Best DNNs output by DRAGON for several time series.

To have a more visual comparison of the different algorithms from the baseline, we used the performance profile as defined by Dolan and Moré (2002). We name $\mathscr{P}$ the set of the 27 datasets, $\mathscr{S}$ the set of the 22 algorithms from the baseline and $l_{p,s}$ the final score (loss) of the algorithm $s \in \mathscr{S}$ on the dataset $p \in \mathscr{P}$. We define the performance ratio $r_{p,s}$ of s on p as:

$$r_{p,s} = \frac{l_{p,s}}{\min\{l_{p,s} : s \in \mathscr{S}\}}.$$

Dataset	Handcrafted	AutoPytorch	AutoARIMA	AutoETS	AutoTheta	StatEnsemble	AutoGluon	DRAGON
COVID	5.192	4.911	6.029	5.907	7.719	5.884	5.805	4.535
Car Parts	0.746	0.746	1.118	1.133	1.208	1.052	0.747	0.745
Electricity Hourly	1.389	1.420	-	1.465	-	-	1.227	1.314
Electricity Weekly	0.769	2.322	3.009	3.076	3.113	3.077	1.892	0.644
FRED-MD	0.468	0.682	0.478	0.505	0.564	0.498	0.656	0.494
Hospital	0.673	0.770	0.820	0.766	0.764	0.753	0.741	0.750
KDD	0.844	0.764	-	0.988	1.010	-	0.709	0.678
M1 Monthly	1.074	1.278	1.152	1.083	1.092	1.045	1.235	1.069
M1 Quarterly	1.658	1.813	1.770	1.665	1.667	1.622	1.615	1.717
M1 Yearly	3.499	3.407	3.870	3.950	3.659	3.769	3.371	3.683
M3 Monthly	0.861	0.956	0.934	0.867	0.855	0.845	0.822	0.900
M3 Other	1.814	1.871	2.245	1.801	2.009	1.769	1.837	2.144
M3 Quarterly	1.117	1.180	1.419	1.121	1.119	1.096	1.057	1.087
M3 Yearly	2.774	2.691	3.159	2.695	2.608	2.627	2.520	2.775
M4 Daily	1.141	1.152	1.153	1.228	1.149	1.145	1.156	1.056
M4 Hourly	1.193	1.345	1.029	1.609	2.456	1.157	0.807	1.155
M4 Monthly	0.947	0.851	0.812	0.803	0.834	0.780	0.782	0.991
M4 Quarterly	1.161	1.176	1.276	1.167	1.183	1.148	1.139	1.190
M4 Weekly	0.453	2.369	2.355	2.548	2.608	2.375	2.035	0.446
NN5 Daily	0.789	0.807	0.935	0.870	0.878	0.859	0.761	0.892
NN5 Weekly	0.808	0.865	0.998	0.980	0.963	0.977	0.860	0.703
Pedestrians	0.247	0.354	-	0.553	-	-	0.312	0.218
Tourism Monthly	1.409	1.495	1.585	1.529	1.666	1.469	1.442	1.434
Tourism Quarterly	1.475	1.647	1.655	1.578	1.648	1.539	1.537	1.471
Tourism Yearly	2.590	3.004	4.044	3.183	2.992	3.231	2.946	2.337
Vehicle Trips	1.176	1.162	1.427	1.301	1.284	1.203	1.113	1.645
Web Traffic	0.973	0.962	1.189	1.207	1.108	1.068	0.938	0.561

Table 2: Mean MASE for each dataset. We did not report all the individual scores from the handcrafted baseline, but the best score from the 15 models for each time series. The grayed values correspond to the minimal loss for the corresponding dataset.

From this we can define the performance profile as the probability for the algorithm $s \in \mathcal{S}$ that the performance ratio on any dataset is within a factor $\tau \in \mathbb{R}$ of the best possible ratio:

$$\rho_s(\tau) = \frac{1}{27}\text{size}\{p \in \mathcal{P} : r_{p,s} \leq \tau\}\,,$$

the function ρ_s is the (cumulative) distribution function for the performance ratio. We compute the performance profile for each algorithm from the AutoML baseline, which can be found Figure 7. From the performance profile, we can see that compared to the baseline,

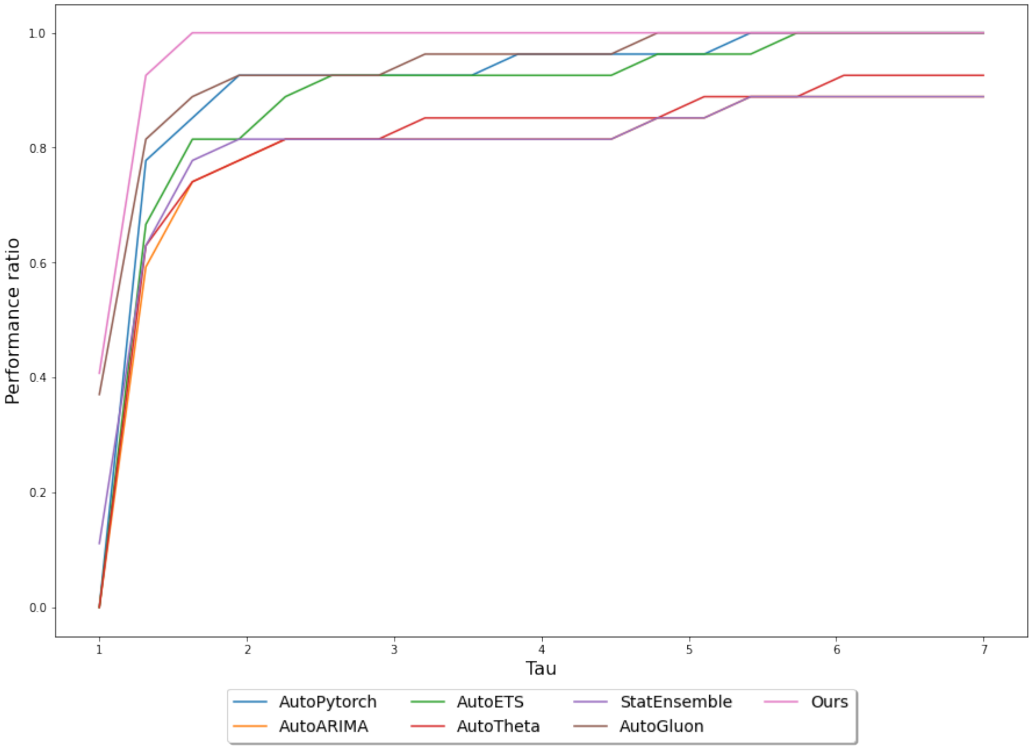


Figure 7: Performance profile $\rho_s(\tau)$ for each algorithm s from the AutoML baseline, with $\tau \in [1, 7]$.

DRAGON has an error close to the best for every dataset. It is also the only algorithm for which the performance ratio is less than two for all datasets. This diagram also suggests that the performance of AutoPytorch and AutoGluon are not that different.

5.5 Computation Time

To be consistent with the other algorithms from the baseline, we set a fixed time budget of 8 hours for our experiments. But in most cases the algorithm found the best solution in less time than this. Figure 8 represents the time convergence of DRAGON for each dataset. For almost every one of them, a close solution to the final one was found in less than an hour. For some datasets like M4 weekly or M1 Quarterly, DRAGON did not improve the results after the first hour. The models from the baseline train faster, with AutoGluon for

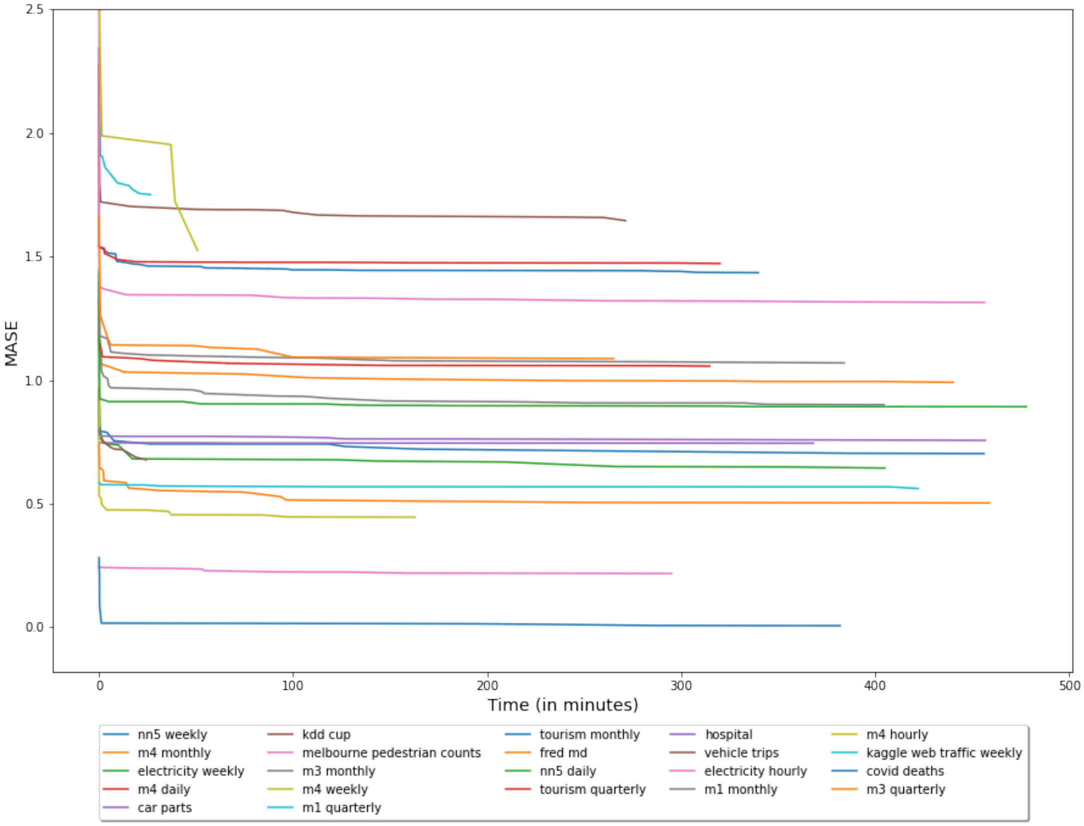


Figure 8: Computation time of DRAGON for each dataset. The curves represent the time when the best loss so far has been found for each dataset.

example having an average runtime of 33 minutes (Shchur et al., 2023). However, those algorithms are based on Machine Learning models, wherease the runtime of AutoPytorch, the other AutoDL framework was set to 4 hours for each datasets. The training time of DNNs are indeed usually higher than for traditional machine learning models. We think we can improve our training time using a multi-fidelity approach. Indeed, with our evolutionary algorithm, every DNN is trained for 100 epochs before being evaluated. With a multi-fidelity approach we could speed up the identification of good performing models and stop training the worst ones sooner.

5.6 Best Models Analysis

In the AutoDL literature, little effort is usually made to analyze the generated DNNs. Shu and Cai (2019) found that architectures with wide and shallow cell structures are favored by the NAS algorithms, which do not generalize well. We performed a light analysis on the best models found by DRAGON to see if our framework favors such structures as well. In this section we try to answer some questions about the results of our framework. To do so, we first define some structural indicatore. We computed them on the best model found for each time series dataset and summarize this in the table 3:

- ***Nodes***: Number of nodes (i.e. operations) in the graph.
- ***Width***: Network width, which can be defined as the maximum of incoming or outgoing edges to all nodes in the graph.
- ***Depth***: The depth of the network, which is the size of the longest path in the graph.
- ***Edges***: The number of edges relative to the number of nodes in the graph. It indicates how complex the graph can be and how sparse the adjacency matrix is.
- The last 7 indicators correspond to the number of occurrences of each layer type within the DNN.

Does DRAGON always converge to complex models or is it able to find simple DNNs? From the table 3 we see that the models found are really small compared to a transformer model for example, and all have less than 8 hidden layers, while we let the algorithms have cells with up to 10 nodes. Moreover, two models consist of only one layer, such as the one displayed Figure 6a. Another indicator of model simplicity is the percentage of feed-forward and identity layers found in the best models. The feed-forward layer (also called MLP Table 3) is the most frequent layer, as it appears on average at least once per graph, although more complex layers such as convolution, recurrence or attention layers are less frequently selected in our search space. This proves that even without regularization penalties, our algorithmic framework does not systematically search for overly complicated models.

Does DRAGON always converge to similar architectures for different datasets? The structural indicators for all datasets from table 3 are significantly different for each dataset, which means that the framework does not converge to similar architectures. As

Dataset	Nodes	Width	Depth	Edges	MLP	Att	CNN	RNN	Drop	Id	Pool
m3 monthly	6	3	4	12	2	1	1	1	0	0	1
covid death	3	1	3	3	1	1	0	0	1	0	0
m3 quarterly	4	2	3	6	1	0	0	2	0	1	0
vehicle trips	4	3	4	8	1	0	0	1	0	0	2
m1 yearly	8	5	5	17	2	2	0	0	1	3	0
m4 monthly	6	3	4	12	2	1	1	1	0	0	1
m3 other	4	4	3	8	2	1	0	1	0	0	0
tourism quarterly	1	1	1	1	1	0	0	0	0	0	0
pedestrian	2	2	2	3	0	0	1	0	1	0	0
nn5 daily	5	5	3	12	1	1	1	0	0	2	0
Web Traffic	7	4	6	16	2	2	2	0	0	1	0
m1 quarterly	1	1	1	1	1	0	0	0	0	0	0
tourism yearly	7	3	6	15	2	1	1	0	1	1	1
electricity weekly	7	5	7	18	3	1	1	0	0	2	0
m4 hourly	5	4	5	11	0	2	1	0	1	1	0
electricity hourly	3	3	3	5	0	1	0	0	0	2	0
m3 yearly	6	4	5	13	1	2	0	0	0	3	0
m4 weekly	2	2	2	3	1	0	0	0	0	1	0
m4 daily	2	2	2	3	0	1	0	0	0	1	0
nn5 weekly	2	2	2	3	1	0	1	0	0	0	0
kdd cup	7	6	5	17	2	1	2	1	1	0	0
hospital	4	4	3	8	1	0	2	1	0	0	0
m1 monthly	2	1	2	2	1	0	0	0	0	1	0
fred md	5	4	3	9	0	1	1	0	1	1	1
car parts	7	3	6	15	2	1	1	0	1	1	1
Mean	4.40	3.08	3.60	8.84	1.20	0.80	0.64	0.32	0.32	0.84	0.28

Table 3: Structural indicators of the best model for each dataset found by DRAGON.

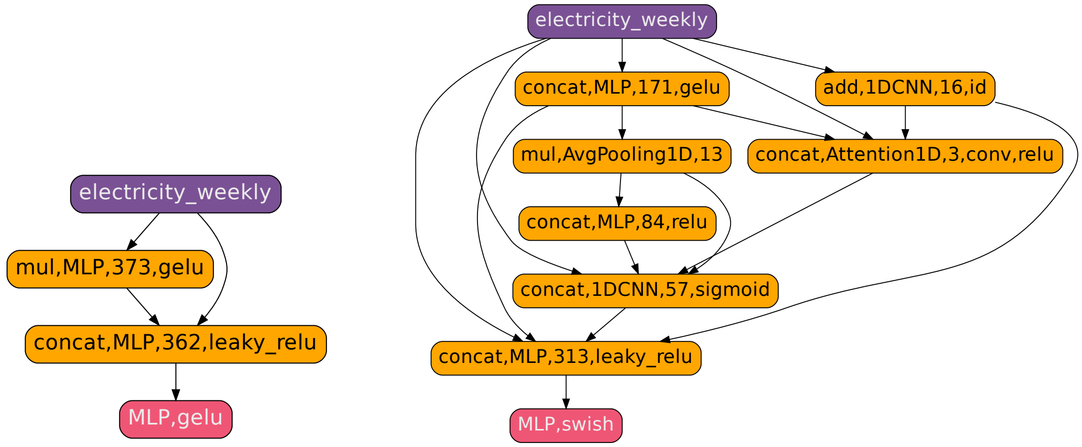
we set the seed, the initial population of size K is identical for each dataset, but then the performance of the evaluated models affects the creation of the following graphs, leading to different final models optimized for each time series. Furthermore, Figure 9 shows that DRAGON can find different performing architectures for the same dataset.

What is the diversity of the operations within the best models?

The MLP layer is definitively the most used operation within the candidates ones. On average, each model from Table 3 is using at least one MLP layer. Interestingly the CNN and Attention layers are more often used than RNN layers, which were designed for time series. Another intersting insight is that every candidate operation has been at least picked once, which states the operations diversity within the best models.

Are the best models still "deep" neural networks or are they wide and shallow as stated in Shu and Cai (2019)?

To answer this question, the observations from Shu and Cai (2019) do not necessary apply to our results. Our models are on average a bit deeper than wide, bearing in mind that the indicators do not take into account the last MLP as shown Figure 5. If we were doing multi-fidelity in the future, this observation might change as one of the reasons mentioned in the paper for wider DNNs is the premature evaluation of architecture before full convergence.



(a) MASE: 0.653

(b) MASE: 0.654

Figure 9: Two different models having similar good performance on the Electricity Weekly dataset (best MASE: 0.644).

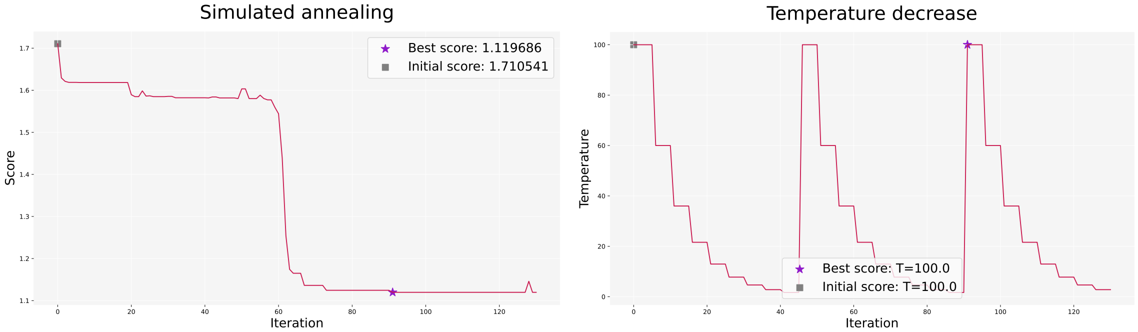
5.7 Ablation Study

We chose two datasets, M1 monthly and Tourism monthly, in order to reduce the number of experiments we had to perform, as the benchmark was quite large. We compared four search algorithms for both datasets; these were random search, a population based evolutionary algorithm (EA) with alternating optimisation of hyperparameters and architecture, as well as a version with joint optimisation, and, lastly, simulated annealing. To explore the search space, we used an exponential multiplicative monotonic cooling schedule in our simulated annealing algorithm: $T_k = T_0.\alpha^k$. We evaluated 40 neighborhood solutions at each iteration to accomplish this. To ensure fairness between each search algorithm, we conducted five experiments with different seeds (0, 100, 200, 300, and 400), and parameterised the algorithms to evaluate 4000 DNNs. The results of this study are presented in Table 4.

Search Algorithm	M1 Monthly	Tourism Monthly
Random search	1.098 ± 0.006	1.645 ± 0.018
EA joint mutation	$\mathbf{1.073 \pm 0.004}$	$\mathbf{1.450 \pm 0.003}$
EA alternating mutation	1.080 ± 0.005	1.451 ± 0.004
Simulated Annealing	1.141 ± 0.044	2.640 ± 0.037

Table 4: Comparison between several search algorithms over two datasets: M1 Monthly and Tourism Monthly. Each configuration has been ran with five different seeds.

The findings suggest that the most exploratory algorithms, specifically the random search algorithm and the evolutionary algorithm, yielded better results than the more locally focused, ie: the simulated annealing. The findings imply the presence of several potential solutions in the search space, but none of them could be accessed by the simulated annealing algorithms from their starting points. The figure 10 indicates that the most effective DNN was achieved with the help of simulated annealing when $Temp = Temp_{max}$. This suggests greater exploration by the algorithm. Additionally, the random search method produced favorable results. The assessment of 4000 solutions for the Tourism Monthly dataset and M1 dataset was completed within 12 minutes and 4 hours respectively, thanks to the parallelisation of the solution. This shows that the search space has been suitably designed for our problem. However, it does not achieve the same level of performance as our evolutionary algorithms, highlighting the significance of our variational operators. Ultimately, both types of mutation produce very similar results.



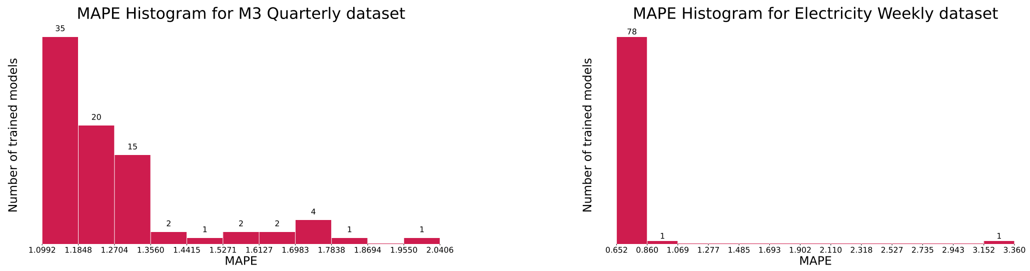
(a) Best score for each iteration of the simulated annealing algorithm.

(b) Temperature value for each iteration of the simulated annealing algorithm.

Figure 10: Simulated annealing algorithm for the M1 Monthly dataset with seed=100, MASE=1.120.

5.8 Nondeterminism and Instability of DNNs

An often overlooked robustness challenge with DNN optimization is their uncertainty in performance (Summers and Dinneen, 2021). A unique model with a fixed architecture and set of hyperparameters can produce a large variety of results on a dataset. Figure 11 shows the results on two datasets: M3 Quarterly and Electricity Weekly. For both datasets, we selected the best models found with our optimization and drew 80 seeds summing all instability and nondeterministic aspects of our models. We trained these models and plotted the MASE Figure 11. On the M3 Quarterly, the MASE reached values two times bigger than our best result. On the Electricity Weekly, it went up to five times worst. To overcome this problem, we represented the parametrization of stochastic aspects in our models as a hyperparameter, which we added to our search space. Despite its impact on the performance, we have not seen any work on NAS, HPO or AutoML trying to optimize the seed of DNNs. Our plots of Figure 11 showed that the optimization was effective as no other seeds gave better results than the one picked by DRAGON.



(a) M3 Quarterly, best MASE: 1.099

(b) Electricity Weekly, best MASE: 0.652

Figure 11: MASE histogram of the best model performances with multiple seeds for two datasets.

6. Conclusion and Future Work

In this article, we introduce DRAGON, a novel algorithmic framework to optimize jointly the architectures of DNNs and their hyperparameters. We initially presented a search space founded on Directed Acyclic Graphs, which is flexible for architecture optimization and also allows fine-tuning of hyperparameters. We then develop search operators that are compatible with any metaheuristic capable of handling a mixed and variable-size search space. We prove the efficiency of DRAGON on a task rarely tackled by AutoDL or NAS works: time series forecasting. On this task where the performing DNNs have not been clearly identified, DRAGON shows superior forecasting capalities compare to the state-of-the-art in AutoML and handcrafted models.

Although we obtained satisfactory results compared to our baseline, we note that our algorithm runs slower than AutoGluon, its main competitor, and does not improve it much. However, we would like to point out that AutoGluon produces mixtures of machine learning models, while DRAGON produces a single DNN. To be more competitive in terms of computation time and results, we could consider using multi-fidelity techniques to identify and eliminate unpromising solutions more quickly, using multi-objective techniques to increase the value of simpler, easier-to-train DNNs, and taking inspiration from AutoGluon and AutoPytorch techniques and blending DNNs and machine learning predictors to further improve forecasting accuracy. Moreover, for each generated architecture, we optimize the hyperparameters using the same evolutionary algorithm. However, hyperparameters play a large role in the performance of a given architecture, and it could be interesting to investigate an optimization that alternates between specific search algorithms for the architecture and for the hyperparameters. In fact, while the graph structure representing the architecture is difficult to manipulate, once fixed, the hyperparameter search space can be considered as a vector that could be optimized with more efficient algorithms such as Bayesian or bi-level optimization, allowing a greater number of possibilities to be evaluated.

Furthermore, given our search space and search algorithms' universality, we could extend our framework to several other tasks. Indeed, only the candidate operations included as node content are task-related, and the representation of DNNs as DAG is not. Further research can test our framework on various learning tasks, necessitating the creation of new operations, such as 2-dimensional convolution and pooling, for the treatment of images, for example. Additionally, this framework can also function as a cell-based search space, utilising normal and reduction cells as opposed to a single convolution operation.

Finally, our study demonstrates that incorporating a variety of cutting-edge DNN operations into a single model presents a promising approach for enhancing the performance of time series forecasting. We consider these models as innovative within the deep learning community, and further research investigating their efficacy could be interested.

Acknowledgments

We would like to thank Margaux Brégère for her careful proofreading and helpful advice. This work has been funded by Electricité de France (EDF). The supercomputers used to run the experiments belong to EDF.

Appendix A. Available Operations and Hyperparameters

Operation	**Optimized hyperparameters**	
Identity	-	
Fully-Connected (MLP)	Output shape	Integer
Attention	Initialization type	[convolution, random]
	Heads number	Integer
1D Convolution	Kernel size	Integer
Recurrence	Output shape	Integer
	Recurrence type	[LSTM, GRU, RNN]
Pooling	Pooling size	Integer
	Pooling type	[Max, Average]
Dropout	Dropout Rate	Float

Table 5: Operations available in our search space and used for the Monash time series archive dataset and their hyperparameters that can be optimized.

Activation functions, $\forall x \in \mathbb{R}^{\mathbb{D}}$

- Id: $\mathrm{id}(x) = x$
- Sigmoid: $\mathrm{sigmoid}(x) = \frac{1}{1+\mathrm{e}^{-x}}$
- Swish: $\mathrm{swish}(x) = x \times \mathrm{sigmoid}(\beta x) = \frac{x}{1+e^{-\beta x}}$
- Relu: $\mathrm{relu}(x) = \max(0, x)$
- Leaky-relu: $\mathrm{leakyRelu}(x) = \mathrm{relu}(x) + \alpha \times \min(0, x)$, in our case: $\alpha = 10^{-2}$
- Elu: $\mathrm{elu}(x) = \mathrm{relu}(x) + \alpha \times \min(0, e^x - 1)$
- Gelu: $\mathrm{gelu}(x) = x\mathbb{P}(X \leq x) \approx 0.5x(1 + \tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$
- Softmax: $\sigma(\mathbf{x})_j = \frac{\mathrm{e}^{x_j}}{\sum_{d=1}^{D} \mathrm{e}^{x_d}} \; \forall j \in \{1, \ldots, D\}$

Appendix B. Monash Datasets Presentation

Dataset	Domain	Nb of series	Multivariate	Lag	Horizon
Carparts	Sales	2674	Yes	15	12
Elec. hourly	Energy	321	Yes	30	168
Elec. weekly	Energy	321	Yes	65	8
Fred MD	Economic	107	Yes	15	12
Hospital	Health	767	Yes	15	12
KDD	Nature	270	No	210	168
M1 monthly	Multiple	1001	No	15	18
M1 quart.	Multiple	1001	No	5	8
M1 yearly	Multiple	1001	No	2	6
M3 monthly	Multiple	3003	No	15	18
M3 other	Multiple	3003	No	2	8
M3 quart.	Multiple	3003	No	5	8
M3 yearly	Multiple	3003	No	2	6
M4 daily	Multiple	100000	No	9	14
M4 hourly	Multiple	100000	No	210	48
M4 monthly	Multiple	100000	No	15	18
M4 quart.	Multiple	100000	No	5	8
M4 weekly	Multiple	100000	No	65	13
NN5 daily	Banking	111	Yes	9	56
NN5 weekly	Banking	111	Yes	65	8
Pedestrians	Transport	66	No	210	24
Tourism monthly	Tourism	1311	No	2	24
Tourism quart.	Tourism	1311	No	5	8
Tourism yearly	Tourism	1311	No	2	4
Traffic weekly	Transport	862	Yes	65	8
Vehicle trips	Transport	329	No	9	30

Table 6: Information about the Monash datasets (Godahewa et al., 2021).

References

Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, and P. Martineau. An exact graph edit distance algorithm for solving pattern recognition problems. In *4th International Conference on Pattern Recognition Applications and Methods 2015*, 2015.

A. H. Aguirre and C. A. Coello Coello. Evolutionary synthesis of logic circuits using information theory. *Artificial Intelligence Review*, 20:445–471, 2003.

A. Alsharef, K. Aggarwal, M. Kumar, A. Mishra, et al. Review of ml and automl solutions to forecast time-series data. *Archives of Computational Methods in Engineering*, pages 1–15, 2022.

F. Assunção, N. Lourenço, P. Machado, and B. Ribeiro. Denser: deep evolutionary network structured representation. *Genetic Programming and Evolvable Machines*, 20(1):5–35, 2019.

G. Bender, P.-J. Kindermans, B. Zoph, V. Vasudevan, and Q. Le. Understanding and simplifying one-shot architecture search. In *International conference on machine learning*, pages 550–559. PMLR, 2018.

A. Brock, T. Lim, J. M. Ritchie, and N. Weston. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017.

A. Camero, H. Wang, E. Alba, and T. Bäck. Bayesian neural architecture search using a training-free performance metric. *Applied Soft Computing*, 106:107356, 2021.

M. Chatzianastasis, G. Dasoulas, G. Siolas, and M. Vazirgiannis. Graph-based neural architecture search with operation embeddings. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 393–402, 2021.

D. Chen, L. Chen, Z. Shang, Y. Zhang, B. Wen, and C. Yang. Scale-aware neural architecture search for multivariate time series forecasting. *arXiv preprint arXiv:2112.07459*, 2021.

K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

J.-B. Cordonnier, A. Loukas, and M. Jaggi. On the relationship between self-attention and convolutional layers. *arXiv preprint arXiv:1911.03584*, 2019.

S. M. J. Dahl. *TSPO: an autoML approach to time series forecasting.* PhD thesis, Universidade Nova de Lisboa Lisbon, Portugal, 2020.

Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

D. Deng, F. Karl, F. Hutter, B. Bischl, and M. Lindauer. Efficient automated deep learning for time series forecasting. *arXiv preprint arXiv:2205.05511*, 2022.

E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91:201–213, 2002.

T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

M. Fiore and M. Devesas Campos. The algebra of directed acyclic graphs. In *Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky*, pages 37–51. Springer, 2013.

R. Godahewa, C. Bergmeir, G. I. Webb, R. J. Hyndman, and P. Montero-Manso. Monash time series forecasting archive. *arXiv preprint arXiv:2105.06643*, 2021.

Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun. Single path one-shot neural architecture search with uniform sampling. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XVI 16*, pages 544–560. Springer, 2020.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

F. Hutter, L. Kotthoff, and J. Vanschoren. *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

R. J. Hyndman and A. B. Koehler. Another look at measures of forecast accuracy. *International journal of forecasting*, 22(4):679–688, 2006.

W. Irwin-Harris, Y. Sun, B. Xue, and M. Zhang. A graph-based encoding for evolutionary convolutional neural network architecture design. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 546–553. IEEE, 2019.

C. Lea, M. D. Flynn, R. Vidal, A. Reiter, and G. D. Hager. Temporal convolutional networks for action segmentation and detection. In *proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 156–165, 2017.

Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

N. Li, L. Ma, G. Yu, B. Xue, M. Zhang, and Y. Jin. Survey on evolutionary deep learning: Principles, algorithms, applications and open issues. *arXiv preprint arXiv:2208.10658*, 2022.

B. Lim, S. Ö. Arık, N. Loeff, and T. Pfister. Temporal fusion transformers for interpretable multi-horizon time series forecasting. *International Journal of Forecasting*, 37(4):1748–1764, 2021.

C. Liu, L.-C. Chen, F. Schroff, H. Adam, W. Hua, A. L. Yuille, and L. Fei-Fei. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 82–92, 2019.

H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017.

H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu. Hierarchical representations for efficient architecture search, 2018a.

H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018b.

Z. Liu, Z. Zhu, J. Gao, and C. Xu. Forecast methods for time series data: A survey. *IEEE Access*, 9:91896–91912, 2021. doi: 10.1109/ACCESS.2021.3091162.

M. Loni, S. Sinaei, A. Zoljodi, M. Daneshtalab, and M. Sjödin. Deepmaker: A multi-objective optimization framework for deep neural networks in embedded systems. *Microprocessors and Microsystems*, 73:102989, 2020.

A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

B. N. Oreshkin, D. Carpov, N. Chapados, and Y. Bengio. N-beats: Neural basis expansion analysis for interpretable time series forecasting. *arXiv preprint arXiv:1905.10437*, 2019.

H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameters sharing. In *International conference on machine learning*, pages 4095–4104. PMLR, 2018.

D. Salinas, V. Flunkert, J. Gasthaus, and T. Januschowski. Deepar: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, 36(3): 1181–1191, 2020.

S. Y. Shah, D. Patel, L. Vu, X.-H. Dang, B. Chen, P. Kirchner, H. Samulowitz, D. Wood, G. Bramble, W. M. Gifford, et al. Autoai-ts: Autoai for time series forecasting. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2584–2596, 2021.

O. Shchur, C. Turkmen, N. Erickson, H. Shen, A. Shirkov, T. Hu, and Y. Wang. Autogluon-timeseries: Automl for probabilistic time series forecasting. *arXiv preprint arXiv:2308.05566*, 2023.

W. W. Shu, Yao and S. Cai. Understanding architectures learnt by cell-based neural architecture search. *CoRR*, abs/1909.09569, 2019. URL `http://arxiv.org/abs/1909.09569`. Withdrawn.

Y. Shu, W. Wang, and S. Cai. Understanding architectures learnt by cell-based neural architecture search. *arXiv preprint arXiv:1909.09569*, 2019.

D. So, Q. Le, and C. Liang. The evolved transformer. In *International Conference on Machine Learning*, pages 5877–5886. PMLR, 2019.

C. Summers and M. J. Dinneen. Nondeterminism and instability in neural network optimization. In *International Conference on Machine Learning*, pages 9913–9922. PMLR, 2021.

Y. Sun, B. Xue, M. Zhang, and G. G. Yen. A particle swarm optimization-based flexible convolutional autoencoder for image classification. *IEEE transactions on neural networks and learning systems*, 30(8):2295–2309, 2018.

E.-G. Talbi. Automated design of deep neural networks: A survey and unified taxonomy. *ACM Computing Surveys (CSUR)*, 54(2):1–37, 2021.

E.-G. Talbi. Metaheuristics for variable-size mixed optimization problems: a survey and taxonomy. *Submitted to IEEE Trans. on Evolutionary Algorithms*, 2023.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

B. Wang, Y. Sun, B. Xue, and M. Zhang. Evolving deep convolutional neural networks by variable-length particle swarm optimization for image classification. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2018.

B. Wang, Y. Sun, B. Xue, and M. Zhang. Evolving deep neural networks by multi-objective particle swarm optimization for image classification. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 490–498, 2019.

C. White, W. Neiswanger, and Y. Savani. Bananas: Bayesian optimization with neural architectures for neural architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 10293–10301, 2021.

C. White, M. Safari, R. Sukthanker, B. Ru, T. Elsken, A. Zela, D. Dey, and F. Hutter. Neural architecture search: Insights from 1000 papers. *arXiv preprint arXiv:2301.08727*, 2023.

L. Xie and A. Yuille. Genetic cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1379–1388, 2017.

C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter. Nas-bench-101: Towards reproducible neural architecture search. In *International conference on machine learning*, pages 7105–7114. PMLR, 2019.

M. Zhang, S. Jiang, Z. Cui, R. Garnett, and Y. Chen. D-vae: A variational autoencoder for directed acyclic graphs. *Advances in Neural Information Processing Systems*, 32, 2019.

G. Zhong, T. Li, W. Jiao, L.-N. Wang, J. Dong, and C.-L. Liu. Dna computing inspired deep networks design. *Neurocomputing*, 382:140–147, 2020.

B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.