Université de Lille

CRIStAL
Centre de Recherche en Informatique, Signal et Automatique de Lille

Inria

EDF

Automated Deep Learning: algorithms and software for energy sustainability

PhD Thesis of the University of Lille
prepared at CRIStAL, EDF R&D and INRIA Paris

Doctoral School n°631 Mathématiques, sciences du numérique et de leurs intéractions (MADIS)
Academic Field: Computer Science

Thesis defended at Palaiseau, on the 24th January 2025, by

Julie Keisler

Committee members:

Marius Lindauer Professor, Leibniz Universität Hannover	Referee
Florian Ziel Professor, Duisburg-Essen Universität	Referee
Carola Doerr Research Director, Sorbonne Université	Examiner
Ludovic Macaire Professor, Université de Lille	Committee president
Margaux Brégère Research Ingineer, EDF R&D, PAST, Sorbonne Université	Supervisor
Claire Monteleoni Research Director, Inria Paris, Professor, University of Colorado Boulder	Supervisor, invited
El-Ghazali Talbi Professor, Université de Lille	Supervisor, invited

PhD Thesis

Université de Lille

Thèse de doctorat

Optimisation de réseaux de neurones : algorithmes et logiciel pour un système électrique durable

Thèse de doctorat de l'Université de Lille
préparée à CRIStAL, à EDF Lab Paris-Saclay et à l'INRIA de Paris

École doctorale n°631 Mathématiques, sciences du numérique et de leurs intéractions (MADIS)
Spécialité de doctorat: Informatique et applications

Thèse présentée et soutenue à Palaiseau, le 24 Janvier 2025, par

Julie Keisler

Membres du jury:

Marius Lindauer Professeur, Leibniz Universität Hannover	Rapporteur
Florian Ziel Professeur, Duisburg-Essen Universität	Rapporteur
Carola Doerr Directrice de Recherche, Sorbonne Université	Examinatrice
Ludovic Macaire Professeur, Université de Lille	Président du jury
Margaux Brégère Ingénieure de Recherche, EDF R&D, PAST, Sorbonne Université	Encadrante
Claire Monteleoni Directrice de Recherche, Inria Paris, Professeure, University of Colorado Boulder	Co-directrice, invitée
El-Ghazali Talbi Professeur, Université de Lille	Directeur, invité

Automated deep learning: algorithms and software for the energy sustainability

Abstract

Current technologies only allow storage by expensive and inefficient means, which makes it difficult to store electricity on a large scale. For the grid to function properly, electricity fed into the grid must match electricity used at all times. Historically, and still today, production resources are planned in advance of demand to maintain this balance. It is therefore crucial to forecast electricity consumption as accurately as possible. The integration of renewable energies, whose production is intermittent and dependent on weather conditions, is making the balance increasingly unstable. Managing this is becoming more complex, making forecasting wind and photovoltaic production now essential.

Statistical learning models are used to make consumption and production forecasts. These models take past values and data from explanatory variables and use them to model the signal. To build efficient models, one must choose the input variables, the type of model, and its parameters. Given the vast number of signals to be forecasted, it would be beneficial to automate these choices to create competitive models. Automated Machine Learning (AutoML) is the process of automating the generation of learning models optimized according to the use case. Over the last ten years, numerous AutoML tools have been developed. However, most of them focus on optimizing classification or regression models on tabular data, or on optimizing neural network architectures for image or text processing. These tools are not appropriate for optimizing electricity consumption and production forecasting models.

This thesis is a progress towards automating the generation of time series forecasting models required for power system management. The research work focused on developing the DRAGON Python package, which offers a range of tools for specific yet widely used models: neural networks. DRAGON can be used to create flexible search spaces encompassing a wide variety of neural networks by simultaneity optimizing the architecture and the hyperparameters. They are encoded by Directed Acyclic Graphs (DAGs), where the nodes are operations, parameterised by various hyperparameters, and the edges are the connections between these nodes. To navigate these graph-based search spaces and optimize their structures, the package proposes various search algorithms based on meta-heuristics and bandits-approaches. This thesis details how DRAGON is used for electricity consumption and production forecasts, enabling state-of-the-art models to be generated for these two industrial use cases.

Optimisation de réseaux de neurones : algorithmes et logiciel pour un système électrique durable

Résumé

Les technologies actuelles ne permettant le stockage que par des moyens coûteux et peu efficaces, l'électricité reste difficile à stocker à grande échelle. Pour le bon fonctionnement du réseau, il est ainsi important qu'à tout instant, l'électricité injectée dans le réseau soit égale à l'électricité consommée. Historiquement et encore aujourd'hui, pour maintenir cet équilibre, les moyens de production sont planifiés par anticipation de la demande; d'où l'importance de prévoir aussi précisément que possible la consommation électrique. Avec l'intégration massive des énergies renouvelables dont la production est intermittente et dépendante des conditions météorologiques, la production devient de plus en plus instable et la gestion de l'équilibre se complexifie : des prévisions des productions éolienne et photovoltaïque sont désormais indispensables.

Les prévisions de consommation et de production sont réalisées à l'aide de modèles d'apprentissage statistique, qui modélisent le signal en se basant sur ses valeurs passées et des données de variables dites explicatives. Pour construire un modèle performant, il est nécessaire de choisir les variables explicatives considérées, le type de modèle ainsi que sa paramétrisation. Au vu du très grand nombre de signaux à prévoir, il pourrait être intéressant d'automatiser ces choix pour créer automatiquement des modèles compétitifs. Le *Machine Learning* automatisé, également appelé AutoML pour *Automated Machine Learning*, est le processus d'automatisation de la génération de modèles d'apprentissage optimisés en fonction du cas d'usage. De nombreux outils d'AutoML ont été développés depuis une dizaine d'années, mais la plupart se concentrent sur l'optimisation de modèles de classification ou de régression sur des données tabulaires, ou sur l'optimisation d'architectures de réseaux de neurones pour le traitement d'images ou de textes. Ils ne sont donc pas forcément adaptés à la prévision de séries temporelles telles que la consommation ou production électrique.

Cette thèse est un premier pas vers l'automatisation de la génération de modèles pour les prévisions des séries temporelles nécessaires à la gestion du système électrique. Les travaux de recherche se sont concentrés sur le développement du *package* Python DRAGON, qui propose divers outils pour optimiser des modèles bien particuliers, mais largement utilisés: les réseaux de neurones. Le *package* rend possible la création d'espaces de recherche plus ou moins flexibles, qui permettent d'englober une grande diversité d'architectures et d'optimiser à la fois l'architecture et les hyperparamètres. Ces espaces de recherche sont encodés par des graphes acycliques dirigés, où les nœuds sont des opérations, paramétrées par divers hyperparamètres, et les arêtes sont les connexions entre ces nœuds. Afin de naviguer dans ces espaces de recherche à base de graphes et d'en optimiser les structures, divers algorithmes de recherche à base de métaheuristiques et de bandits sont proposés dans le *package*. Après une présentation de DRAGON, cette thèse détaille comment ce *package* est utilisé pour les prévisions de consommation et de production électrique et permet de générer des modèles à l'état de l'art dans ces deux cas d'usage industriels.

Remerciements

La thèse est un peu plus qu'une expérience scolaire ou professionnelle. C'est également un projet personnel dans lequel l'étudiant s'investit pendant trois ans. A partir d'un sujet initial, il revient au doctorant de produire et créer le contenu qu'il ou elle juge le plus pertinent pour la communauté scientifique et, dans le cas d'une CIFRE, pour l'entreprise. Même si ce parcours se fait en majorité seul, beaucoup de personnes sont présentes comme support tout au long du chemin.

Une thèse n'est rien sans un projet de recherche initial. J'ai eu beaucoup de plaisir à travailler sur mon sujet pendant trois ans et j'aimerai remercier El-Ghazali et Sandra de m'avoir fait confiance pour le traiter. Sandra, j'espère que les outils développés au sein de ma thèse te seront utiles pour continuer à populariser les réseaux de neurones au sein de l'équipe. Gilles, merci pour ton accueil à EDF, même avant le début de ma thèse. Merci pour ta bienveillance, pour toutes tes petites attentions, ton petit soldat trône fièrement sur l'étagère de mon salon. *Then there are the supervisors who have jumped on the bandwagon. Claire, thank you for welcoming me to your lab, which was so well placed at the time, just a 5-minute walk from my apartment. I am appreciative of the kindness, advice, encouragement, and support you have shown me; they have been deeply meaningful. I am honored by your confidence in my abilities. I have enjoyed our conversations, both scientific and personal, and I am eagerly anticipating our future collaborations.* Enfin Margaux, si je devais énumérer toutes les choses pour lesquelles je souhaite te remercier il faudrait rajouter une dizaine de pages à ce manuscrit qui est déjà bien long. Je me contenterai de mentionner ton écoute, ton soutien infaillible pendant ces trois ans et toutes les heures passées à corriger les doubles parenthèses de mes papiers (en voici une pour la route)). J'espère avoir récupéré un peu de tes capacités en modélisation mathématiques, de ta rigueur et de ton intégrité humaine et scientifique. Passer d'une relation d'amies proches à une relation d'encadrement n'est pas du tout évident, et ça a été finalement l'une des meilleures expériences de ma thèse.

I would also like to express my sincere gratitude to the members of the jury. Florian and Marius, I appreciate your thorough engagement with my manuscript and your candid assessment of its strengths and shortcomings. I had the privilege of meeting all three of you, along with Carola, during my thesis at conferences. I greatly appreciated your willingness to share your insights on my research work, and I am grateful for your participation in this jury. Merci également à Ludovic d'avoir proposé de faire partie du jury,

pour son écoute attentive, son soutien et son aide pour l'organisation de la soutenance.

J'ai passé beaucoup de temps à EDF durant mes trois années de thèse, et ça a été l'occasion d'échanger et de travailler avec beaucoup de monde. En particulier les anciens ou actuels membres de l'équipe R39, le "meilleur groupe de la R&D", c'est toujours un plaisir de vous retrouver à la pause café. Adnane, Amaury, Bachir, Christian, David, Elaine, Elise, Guillaume L, Hugo, Isuru, Lionel, Manel, Mariem, Quentin, Thi Thu, Véronique, Virgile, Yann, je n'aurai pas la place d'écrire quelque chose pour chacun d'entre vous mais vous avez tous marqué mes trois années de thèse. Un merci tout particulier à Yannig, le phare de la R&D qui a bien illuminé mon chemin depuis notre tout premier entretien bien avant le début de ma thèse. C'est toujours l'esprit apaisé et des idées plein la tête que je ressors de nos échanges. Merci à Ikrame d'avoir plaidé ma cause pour mes requêtes plus ou moins farfelues telles que partir en conférence à l'autre bout de l'Europe en train. Sur une note moins professionnelle, je tiens à remercier Félicie, ma chief happiness officer, avec qui j'ai eu l'occasion de découvrir tout ce que le CSE d'EDF avait de mieux à offrir, en compagnie d'Anatole et Théo. C'est toujours un plaisir de te croiser, toi et ton sifflet, au bureau comme ailleurs. Tu vas nous manquer durant ces longs mois de voyage, dont j'ai hâte d'entendre tous les détails. Merci à Yvenn aussi, évidemment, pour pleins de choses et notamment pour l'opportunité de donner mes premiers cours. J'espère, tout comme toi, qu'on pourra continuer à collaborer ensemble, professionnellement mais aussi sur le terrain ! En dehors de R39, j'ai eu l'occasion de côtoyer l'équipe R38, merci notamment à Abdelhadi de croire en le potentiel de ma thèse pour l'éolien, ça a été un plaisir de travailler avec toi. Merci à Boutheina également, tu as énormément de choses à m'apprendre et j'ai très hâte de recommencer à travailler avec toi. Merci également aux collègues de Sequoia avec qui j'ai pu échanger quelques ballons à l'urban d'Orsay mais aussi quelques travaux, je pense notamment à Ghislain et Tahar. Enfin il y a les collègues qui deviennent des amis, et des amis qui deviennent collègues. Eva et Louise, ça a été un plaisir de vous croiser régulièrement dans les locaux de la R&D.

Cette thèse a également été l'occasion de croiser pleins de doctorants, à EDF, à l'INRIA, à l'université de Lille mais aussi en conférence. Etienne, le meilleur conteur de la R&D, aussi impressionnant en dribble qu'en écriture d'articles, merci pour notre petite collab sur l'éolien qui nous a menée jusqu'à Vienne. En parlant de longs voyages, merci à Madina et Thomas de m'avoir accompagnée dans ces longues heures de train pour le fin fond de l'Italie. Bari et Syracuse resteront certainement les plus beaux voyages de mes années de thèse, et j'ai été ravie de les partager avec vous. Madina, j'ai été ravie de te voir arriver à Saclay en septembre dernier, je sais que je laisse DRAGON entre de bonnes mains. Petite pensée également à tous ceux qui ont partagés mon bureau à Palaiseau : Joseph, Nathan, Eloi, Guillaume, Caroline et Maria Camilia. Pour les cinq derniers, ainsi que pour Nina et Bianca qui forment avec moi la dream team de Margaux, bon courage pour la fin de vos thèses ! *Thank you to the members of the ARCHES team, both students and researchers, for welcoming me into your team. Anastase, Clément, David,*

Emmanuel, Guillaume, Graham, Maya, Renu, with you I discovered life in a research lab, and I liked it so much that I'm staying! Enfin, cette thèse à été l'occasion d'encadrer trois excellents stagiaires, Alban, Keshav et Roxane. Je vous souhaite tout le meilleur pour vos projets futurs.

Si l'encadrement professionnel est important pour une thèse, l'entourage personnel l'est tout autant. Le mien est exceptionnel et m'a permis de garder le moral, notamment dans les moments stressants et difficiles. Merci à Marion et Babeth tout particulièrement, vous avez été des piliers essentiels durant ces trois années, et surtout en juillet 2024 ! Merci à toute l'équipe Obédix et affiliés, Agathe, Amélie, Aurore, Chloé, Gaspard, Geof, Dom, Fanny, Hiba, Juliette, les Louis, Maxime, Oumar, Raph, Victor et à Aël et Zac qui auraient bien fait de nous rejoindre dès le début. Promis cette année au ski je ne serai pas en train de lancer des codes sur mon PC au lieu d'aider à préparer le goûter. Merci à Margaux et Amaury grâce à qui j'ai découvert le fantastique monde de l'électricité. Merci à Noémie, pour nos créations musicales et nos soirées rhum/twister loufoques et à Noëmie pour celles chez papy. Merci à Charlotte et Guillaume avec qui les fous rires nerveux de la prépa reviennent toujours et à Julie et Camille avec qui j'ai pu fêter dignement le rendu de mon manuscrit. Merci enfin à toutes les fooballeuses que j'ai pu croiser que ce soit avec PSL, le CA Paris ou Paris IX, le foot a été un vrai moyen d'évacuer tous ces algorithmes de ma tête. Petite pensée particulière pour Camille et Inès qui me suivent depuis un moment, j'ai bien moins de motivation à braver le froid et la pluie quand je vous sais absentes de l'entraînement.

Enfin, je tiens à remercier ma famille et ma belle-famille. Merci à Karine et Samuel pour votre accueil parmi vous et votre soutien. Ayant tous les deux fait des thèses c'est en partie en voyant votre parcours que je me suis dit que ça pourrait m'intéresser. Aziliz, Bleuenn, Margot et le petit dernier, Martin, revenez quand vous voulez envahir notre appartement. Je suis extrêmement fière de pouvoir présenter mes travaux devant mes grands-parents, venus (certains pour la première fois !) jusqu'à Paris pour ma soutenance. Merci à tous les quatre de faire ce déplacement. Merci à mes parents, de m'avoir fortement poussée vers la voie scientifique. Je vous en voulais peut être un peu à l'époque, mais finalement ça s'est révélé être un très bon conseil. Merci à vous deux, mais aussi à Lisa, de vous intéresser à mes travaux, même si vous ne comprenez pas toujours tout. Merci d'être toujours si fiers de moi. Votre fierté est pour moi une source de motivation inégalable. Enfin, ce manuscrit n'aurait probablement jamais vu le jour si tu n'avais pas été là. Merci, Elisa, pour ton affection, ton soutien dans tous mes projets et pour tout le bonheur que tu m'apportes au quotidien. Merci d'avoir su comment me soutenir dans les moments de doute et d'avoir su quand célébrer mes succès.

Symbols

The manuscript uses the following symbols throughout, with additional notations occasionally defined within specific chapters or sections.

Domain	Symbol	Description
General	i, j	Samples index
	$\mathcal{D}$	Dataset
	$\mathcal{D}_{\text{train}}$	Training set
	$\mathcal{D}_{\text{valid}}$	Validation set
	$\mathcal{D}_{\text{test}}$	Test set
	X	Input features
	Y	Output target
	f	Machine learning model
	t	Timestamp index
	N	Number of samples
	H	Number of daily instants
	F	Number of input features
Automated Machine Learning	ℓ_{train}	Training Loss function
	ℓ	AutoML Loss function
	T	Search Algorithm budget
	K	Search Algorithm population
	Ω	Search space
	λ	Set of hyperparameters
	Λ	Hyperparameters search space
	α	Neural network architecture
	$\mathcal{A}$	Architectures search space
	δ	Neural network weights
	Δ	Weights search space
	Γ	Graph encoding a neural network

Contents

List of Figures

List of Tables

Part I

Research Context

Chapter 1

Introduction

The objective of this thesis is to propose methodologies and tools for optimizing deep neural networks employed for load consumption and wind production forecasting. In this chapter, we first present the industrial context motivating this work: the need for many forecasting models to manage electricity systems. Then, we introduce the need for tools to automatically create effective models for forecasting tasks and present the field of Automated Machine Learning (AutoML). We finally detail how AutoML has been used to tackle the energy forecasting task. The rest of the chapter is devoted to the description of our contributions.

Contents

1.1 Industrial Context

1.1.1 The Electricity Sector

1.1.1.1 Presentation of the electrical system

In this thesis, we study the electricity sector. Electricity is still difficult to store at large scale. The existing storage methods such as batteries, pumping or hydrogen remain inefficient or expensive and have a limited capacity. Moreover, at any given time, the amount of electricity fed into the grid must match the amount of electricity consumed. Therefore it is essential for the electrical network to maintain a strict balance between load and production. If the imbalance is too great, the system runs the risk of blackout.

The electricity system is expected to be reliable. It means the customers wish to be supplied with the energy requested with a very high degree of confidence. In other words, when someone turns on a light switch, they assume the signal will come on immediately. Thus, historically, and still today, the balance between load and production is maintained by first forecasting the load and then adjusting the production resources accordingly. It is therefore crucial to forecast electricity consumption as accurately as possible.

To meet the challenges of climate change, European countries are planning to invest heavily in renewable energies to decarbonise the electricity sector. For example, wind power generation in Europe is expected to increase from 204 GW per year in 2022 to more than 500 GW in 2030[1]. Some of these renewables, such as wind and solar photovoltaic (PV), are intermittent: their production depends on weather. Therefore, their integration increases uncertainty on the production side and requires accurate forecasts for production in addition to load.

In the event of an imbalance, the Transmission System Operator (TSO), which is responsible for balancing supply and demand at national or regional level, must call on reserve resources to compensate for over- or under-production. These reserve resources are flexible power plants whose production can be changed at short notice. They are potentially large emitters of greenhouse gases. This is why good forecasting is essential when integrating renewable energies to achieve energy sustainability.

1.1.1.2 Electricity market

Maintaining this balance is all the more complex given that electricity production and supply depend on a competitive market. Since February 1997, the European electricity market has been gradually liberalised at the request of the European Union. Prior to that date, the markets in most European countries were non-competitive and organised around four major monopolies: generation, transmission (high-voltage lines), distribution and sales to businesses and consumers. Prices were set nationally by governments or regulators, and customers could not choose their electricity supplier. Following market

[1] https://ec.europa.eu/commission/presscorner/detail/en/ip_23_5185

liberalisation, only the transmission (managed by TSOs - Transmission System Operators) and distribution (managed by DSOs - Distribution System Operators) parts of the market remained monopolised. In France, these are managed by RTE and ENEDIS respectively.

Production and sales, on the other hand, have become commercial activities involving all kinds of players, such as independent producers, traders, brokers, etc. The various players in the market draw up contracts for the supply or production of electricity over various time horizons in order to maximise financial gains. Each time horizon corresponds to a specific market with its own rules. The forward market is the long-term market, for energy sales or purchase contracts with a horizon of up to several years. The spot market is the day-ahead market. This is the central market that sets the "final" price of electricity, hour by hour, according to supply and demand. The intra-day market is used to correct schedules in the event of, for example, a generation system failure or a bad weather forecast. The near real-time balancing market allows the TSO to ensure a balance between generation and load at each time step.

The contracts will dictate what producers must inject into the grid and what suppliers' customers must consume. Producers and suppliers are responsible for guaranteeing their own balance. The TSO is responsible for maintaining the global balance, calling on reserve resources in the event of imbalances and imposing financial penalties on producers or suppliers who are responsible for imbalances.

The market is common to all European countries, allowing international contracts and physical exchanges of electricity to take place. While prices vary by country, they are often significantly affected by load or production in neighboring countries. A typical example is Germany's overproduction of wind power during periods of very favourable wind conditions. To maintain its balance, it sells it surplus of electricity to its neighbors. These countries will see their spot prices fall as a result of this influx of electricity. Local producers of energy that is more expensive than wind power will be forced to turn off some of their power plants.

1.1.1.3 Electricité de France

This thesis was funded and carried out in partnership with EDF (Électricité de France). EDF is the main producer and supplier of electricity in France and one of the largest electricity company in Europe and the world. It produces electricity from a diverse mix of sources, including nuclear, renewable and fossil fuels. Prior to the liberalisation of the electricity market in France, EDF operated as a monopoly. It was the sole provider of electricity in France, controlling every aspect of generation, transmission, distribution and supply. It was therefore essential to have forecasting tools for both load and production before the liberalisation of the market, and this need has only increased since.

1.1.2 Forecasting for the energy sustainability

1.1.2.1 Various forecasting tasks

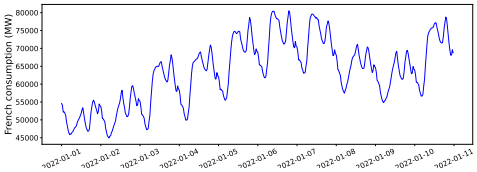
With this brief introduction to the electricity sector, we begin to see the different (and numerous) load and renewable production forecasting needs of the various market players such as EDF. These needs are first and foremost a means of ensuring the smooth operation of the network in a context of increasing electrification and the massive introduction of renewable energies, but they are also a tool of financial optimization for the various players. In addition, the accelerating flow of data, and its greater reliability and precision at ever finer temporal and spatial scales, is enabling the emergence of more complex and accurate learning models, such as machine learning models or neural networks, which were hardly ever used before.

In this thesis we focus on load forecasting and wind energy forecasting. The different market participants (generators, traders, TSOs, DSOs, municipalities, etc.) have different needs for forecasts to ensure the smooth operation of the grid (especially for TSOs and DSOs) or to optimize trading strategies. These forecasts may have different horizons, different levels of aggregation and be applied to different geographical areas.

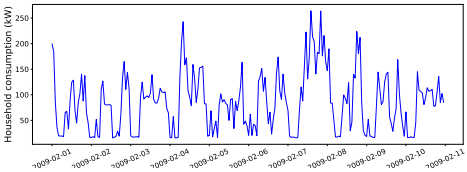
Forecast horizons can range from nowcasting, e.g. hourly forecasts, to several years. Short-term forecasting helps to schedule production and make offers on the intraday or spot markets. Longer horizons will be crucial to plan power plants maintenance or to intervene on the futures market. Very long-term production and load trends (several decades) are also of interest for projects involving the construction or renovation of large production facilities such as offshore wind farms, dams or nuclear power plants. Statistical forecasting methods are mainly used for short horizons but the very long ones are made using modelling methods.

Secondly, the level of aggregation depends on the need. Regarding the load forecast, electricity suppliers such as EDF have more or less extensive customer portfolios, resulting in different levels of aggregation of electricity load. On a larger scale, TSOs are interested in the total load of their perimeter, which may represent the entire national load, as is the case for RTE in France, or a large region, as is the case for the German TSOs Amprion, EnBW, Tennet and 50Hertz Transmission, which divide the country into four regions. Similarly, for electricity production, the aggregation depends on the amount of wind farms an owner has. A wind farm owner needs to know her farm's production in order to make offers to the market at different horizons, depending on whether she wants to intervene in the spot or futures market. The TSO needs forecasts for all the intermittent energy produced in its area. These forecasts, added to those for consumption, make it possible to anticipate potential imbalances that could arise and call up sufficient reserve resources to ensure the stability of the grid.

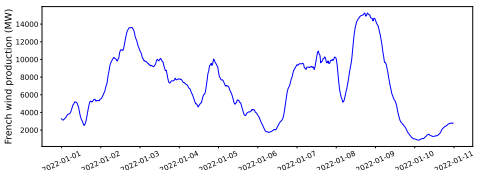
Examples of electricity signals are shown in Figure 1.1.



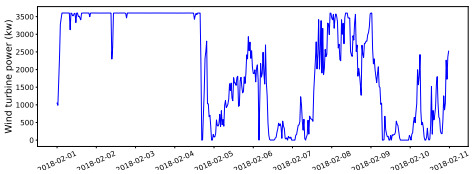
(a) Hourly french load in January 2022.



(b) Hourly french household load in February 2009.



(c) Hourly french wind power in January 2022.



(d) Hourly turkish turbine production in January 2018.

Figure 1.1: Load and wind production signals at various aggregation levels

1.1.2.2 Modelling

Let's formalize the forecasting setting. For each time step t, a value $X_t \in E$ containing various explanatory variables useful for the forecast (historical loads, weather variables, economical factors) is associated to a target time series realization $Y_t \in F$. For $(X_t, Y_t) \in E \times F$, the forecasting problem aims to find a forecasting model f such that:

$$E \to F$$
$$f : X_t \mapsto f(X_t) = \hat{Y}_t \,,$$

where $\hat{Y}_t \in F$ will be the forecasted value of Y_t by the model f. In the case of load and renewable production forecasting, the ensembles E and F can be of various kinds, and the model f depends on those ensembles. In what follows, we focus on point forecast, namely $F \subset \mathbb{R}$. In this case, the forecasted value Y_t might be the load of a given set of consumers or the power generated by a given set of power plants at time t.

Time series models Both load and production can be seen as a time series. Thus Y_t can be predicted using one of the last known values. In this case we can write $X_t = Y_{t-\tau}$, where $\tau \in \mathbb{N}^\star$. Typically, τ might correspond to a couple of hours or days, and therefore $E = F \subset \mathbb{R}$. The model f can be restricted to the identity, namely $\hat{Y}_t = X_t = Y_{t-k}$, to get a persistence model. Despite its great simplicity, such a model can be difficult to beat in certain applications, especially when the forecast horizon modeled by τ is short or when the level of aggregation is low as mentioned by Siebert (2008). Persistence modeling is regularly used as a baseline in the literature.

The longer the forecasting horizon, the less information from a single point will be sufficient to forecast the target value. In order to learn signal patterns, seasonality and trends, time series forecasting models were developed based on several past time steps. In this case, if $n \in \mathbb{N}^\star$ is the number of input values, $E = F^F$. The simple moving average model: $\hat{Y}_t = \frac{1}{F}\sum_{i=k}^{F+\tau} X_{t-i}$ and its variants such as the Autoregressive Integrated Moving Average (ARIMA) model (Amini et al., 2016; Juberias et al., 1999) can be used to model the signal. Another popular but simple statistical model is the exponential smoothing, which is a weighted sum of past observations with exponentially decreasing importance of past data (see Taylor 2008 or Nowicka-Zagrajek and Weron 2002 for applications to load forecasting and Milligan et al. 2003 for an application to wind power forecasting).

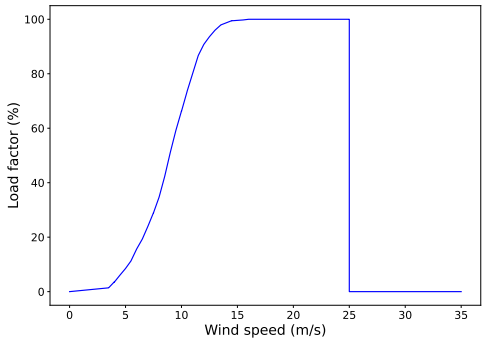
More recently, machine learning and deep learning models have been used to model the complex relationship between historical time steps and current values. For instance, Recurrent Neural Networks (RNN) such as Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) have been designed to treat time series data and have been used extensively for electricity forecasting (see Almalaq and Edwards 2017 for applications to load forecasting and Shi et al. (2018) for applications to renewable energy forecasting). Within the deep learning community, there has been a growing interest in attention-based models such as the Transformer model (see e.g., Vaswani et al. 2017), due to their significant impact on Natural Language Processing (NLP) tasks. Naturally, they have also been tried on electrical time series (see for example Novaes et al. 2021 or Jin et al. 2021).

Time series modelling is a good starting point to forecast load and production data, but the complexity of these signals lies in the fact that although they are time series, they are strongly influenced by external variables. Therefore forecasting their future values can also be seen as a regression problem. This is even truer as the forecast horizon is growing. Past data becomes then less and less sufficient to obtain good forecasts. Furthermore, in the industrial context, completely removing past data from the set of explanatory variables can help improve the models interpretability and ensure good forecasts even in the case of data stream issues.

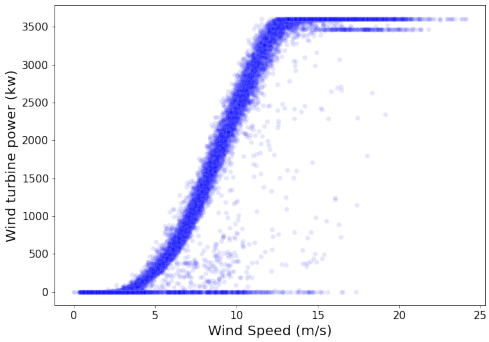
Explanatory Variables Load and production signals are influenced by a lot of external variables. For renewable energy forecasting, weather conditions play a big role in explaining the target. For example, the power generated by a wind turbine depends directly on the wind speed. The relationship between the two can be described by a power curve. The physical theoretical formula is given by (Siebert, 2008):

$$Y_t = f(X_t) = \frac{\rho_{air}}{2} c_p \times A_r \times X_t^3 \,,$$

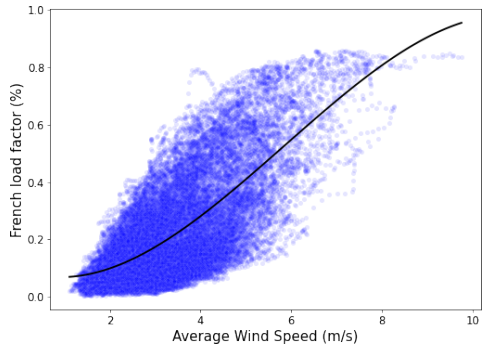
where $Y_t \in \mathbb{R}^+$ is the turbine power, ρ_{air} is the air density, A_r is the area swept by the turbine, c_p is the turbine power coefficient and $X_t \in \mathbb{R}^+$ is the wind speed at time t at



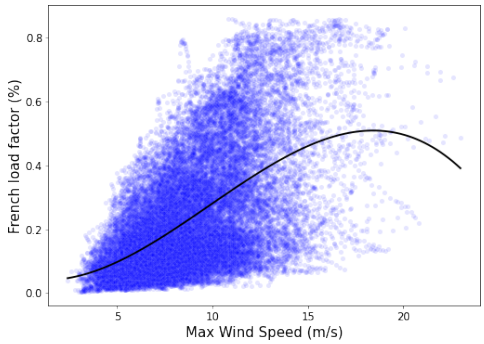
(a) Theoretical power curve.



(b) Hourly Turkish wind turbine power as a function of the wind speed at the turbine level.



(c) Hourly French wind load factor in 2021 and 2022 as a function of the French average wind speed at 100 meters.



(d) Hourly French wind load factor in 2021 and 2022 as a function of the French maximum wind speed at 100 meters.

Figure 1.2: Wind load factor as a function of wind speed. For the Figures 1.2c and 1.2d, a least squares polynomial regression of order 3 has been fitted.

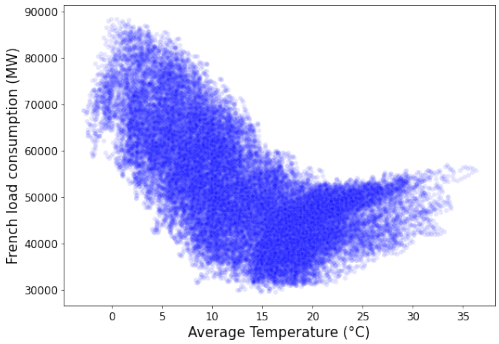
hub height. In practice, it is difficult to observe a weather variable. It is then replaced by the forecast from a given weather model with a given forecast horizon. This function is only valid up to a certain value of maximum wind speed, at which point the turbine is stopped to prevent damage. Figure 1.2a shows the typical shape of this function. Looking at a real signal as in Figure 1.2b, where the hourly power production of a Turkish turbine[2] is plotted as a function of the hourly wind speed value at turbine level, the theoretical power curve appears, although many points remain at zero for any wind speed. This may be due to the shutdown of the plant for economic or maintenance reasons. At the scale of French wind power production, a similar form can be found by plotting the French hourly wind load factor as a function of the French mean wind speed at 100 metres, as shown in Figure 1.2c, and the French maximum wind speed, as shown in Figure 1.2d.

Weather variables also influence the load. Temperature, cloud coverage or wind speed for instance may have an impact. Among them, temperature is usually one of the most important due to heating and cooling equipment. The relationship between French national load and temperature is depicted in Figure 1.3a. The typical pattern is an increasing load when the temperature drops, due to the large number of electric heaters in France, followed by a certain minimum as the temperature falls, before rising again due to air conditioning. This dependence still exists at lower levels of aggregation, but is less pronounced. For example, the dependence on local temperature of the dataset Hebrail and Berard (2006), which contains the individual load of a house in the south of Paris between 2006 and 2010, is shown Figure 1.3b. It is more difficult to find a relationship between the temperature and the load signal.

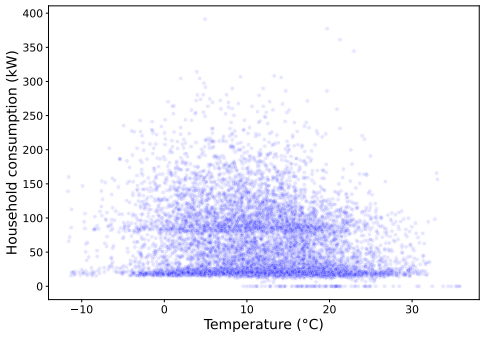
Lifestyles also influence the load, which can therefore vary according to various calendar variables such as the month, day of the week, school holiday periods, etc. On a national scale, for example, there is a marked difference between weekdays and weekends, as shown Figure 1.3c. Industries shut down at weekends, which explains this drop in load. On the other hand, at the individual level, rest days show the highest load, as shown Figure 1.3d. In this figure, in addition to the duality of weekdays and weekends, we can see two days that show a sort of in-between pattern: Wednesday and Friday. Weekdays and temperature are only two examples of the explanatory variables that may influence the load. In practice, the useful explanatory variables depend on the type of consumers, the geographical area or the level of aggregation.

Regression Models The complexity of forecasting electrical signals lies in the fact that they are halfway between time series and regression. Pure time series models are applicable in some cases, but can be limiting in others if they do not include explanatory variables. Regression models often assume independent and identically distributed data, which is not really the case for these signals, even if the historical target data is not included in the input vector.

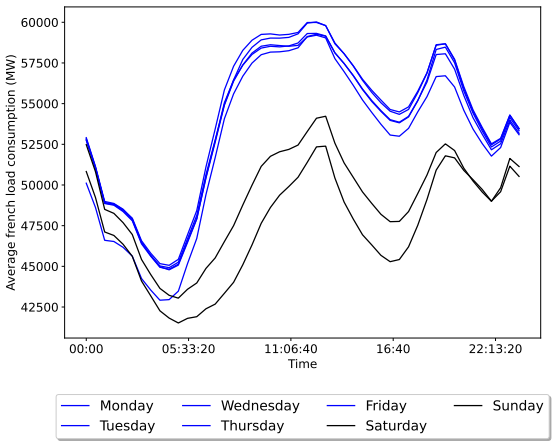
[2] https://www.kaggle.com/berkerisen/wind-turbine-scada-dataset



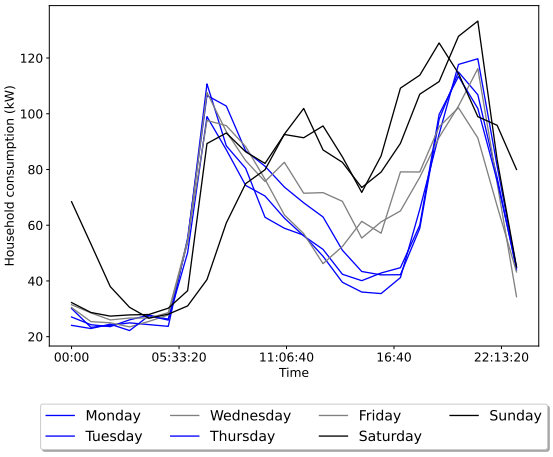
(a) Hourly french load in 2021 and 2022 as a function of the average temperature in °C.



(b) Hourly french household load in Sceaux (south of Paris) during the year 2009 as a function of the local temperature in °C.



(c) Average hourly french load in 2021 and 2022 by weekday.



(d) Average load of a french single household in Sceaux (south of Paris) during the year 2009 by weekday.

Figure 1.3: Effects of the temperature and the weekday on the load at various levels of aggregation (national and individual levels).

Statistical and machine learning techniques have been developed to capture these complex dependencies. These range from ARIMAX models (Hong and Fan, 2016), support vector machines (Mohandes, 2002), random forests (Dudek, 2015) or gradient boosting algorithms (Taieb and Hyndman, 2014), as reviewed by Zhang et al. (2021), to deep learning models (Almalaq and Edwards, 2017; Massaoudi et al., 2021). Typically, dependencies on past data as well as explanatory variables can be modelled using architectures that mix recurrent and convolutional layers.

In the case of load forecasting, to assume that the load signal is a sum of effects depending on external variables is an efficient way of capturing the dependencies on them. The models used are therefore mostly linear regressions, where each coefficient is applied to a (possibly) non-linear function of a given explanatory variable or a subset of explanatory variables. These non-linear functions may be splines in the case of generalised additive models (GAMs), applied to different explanatory variables, crossed or not. GAMs have been introduced for the load by Pierrot and Goude (2011) and Fan and Hyndman (2011). They have been very successful for load forecasting and won a number of competitions (see, for example, Nedellec et al. 2014 and Farrokhabadi et al. 2022). Some works such as Nagy et al. (2016) has been done to apply them to wind power forecasting.

For models functioning almost as pure regression models, the impact of recent load forecasts can be incorporated via calibration models. These treat the model forecasts as a time series and modify them using past data to correct the model. These two-stage models are a way to efficiently treat the forecasting of electrical signals as a hybrid time series and regression task. Recalibration methods can range from auto-regressive (AR) corrections to more complex algorithms. For example, Ba et al. (2012a) proposed an online learning algorithm to track the smoothing functions of additive models and thus adapt load models to an ever-changing environment, and Vilmarest (2022) studied the use of Kalman filters to recalibrate various machine models for load forecasting.

More complex settings For all the works mentioned above, the target variable $Y_t \in \mathbb{R}$ is forecast from an input vector $X_t \in \mathbb{R}^F$. But recently, new formats of data have been explored, such as satellite images (for example, Jasiński (2019) used nightlight images to forecast the load of an area of Poland and Le Guen (2021) forecast short-term solar energy with fisheye images) or weather grids, to incorporate a spatial dimension into weather data. Weather grid data such as Numerical Weather Prediction (NWP) data may be used in the same way as image data where each channel represents a weather variable. del Real et al. (2020) used the weather grids provided by the French meteorological system to forecast French national consumption and Bosma and Nazari (2022) demonstrated the interest of using numerical NWP grids to forecast wind and solar production in California. To handle these spatio-temporal datasets, several deep learning models, mostly based on convolutions, have been studied. Finally, in the case of load forecasting, even more original input formats have been tried, such as text data

(Obst et al., 2024; Bai et al., 2022) or mobility data (Doumèche et al., 2023).

Similarly, the output data can be more complex than a scalar. For example, several time steps can be predicted at once. This can be useful for the specific application (e.g. knowledge of the evolution of the time series), but it can also create more robust models or help to capture longer term dependencies. For probabilistic forecasting, multiple outputs might also be used. In the case of point forecasting, a single value - the mean - is output. But in order to have more information on the model uncertainty, probabilistic forecasting approaches have been developed where several quantiles can be forecast instead of just the mean. Probabilistic forecasting was investigated by de Vilmarest et al. (2023) for GAMs applied to probabilistic load forecasting and Yu et al. (2019) for hybrid neural networks applied to probabilistic regional wind power forecasting. The quantile forecasting can be done with one model per quantile or with a multi-output including all quantiles. Finally, multivariate forecasting might also be of interest for both load and production forecasting, to simultaneously forecast the consumption of several households or regions, or the production of various turbines or power plants. Graph Neural Networks (GNNs) are particularly well suited to treat this type of data (see for example Khodayar and Wang 2018, Khodayar et al. (2021) or Campagne et al. 2024 for applications to wind power and load forecasting respectively).

In summary, the electric system relies on a tremendous number of forecasting problems. The forecasts are based on different input and output data formats, which means that the models used can vary significantly from one application to another. Furthermore, any improvement in forecasting has a significant financial impact. To meet this need for high-performance models for various forecasting problems, tools must be developed to automatically generate models for any given forecasting task. The field of research surrounding the automated creation of efficient machine models is called Automated Machine Learning (AutoML). In the context of energy forecasting, AutoML can be a solution for finding performing models more efficiently and with less human input.

1.2 Automated Machine Learning (AutoML)

Automated Machine Learning (AutoML) is a rapidly evolving field of research with the objective of automating machine learning pipelines, which typically consist of several components, including data processors, feature extractors, and machine learning algorithms. A machine learning expert must explore many potential choices to find the right components for a specific task (e.g., image classification, time series forecasting, text generation). The right choices are made through experience, intuition, and experimentation. Designing a good machine learning pipeline for a given task is challenging for non-experts and experts new to the task, and it remains so regardless of expertise level.

Two main sub-problems are Hyperparameter Optimization (HPO) and model selection. Hyperparameters must be optimized for a machine learning model that has

already been chosen. In contrast, model selection entails choosing the most suitable machine learning model with pre-determined hyperparameters, which are optimized for each candidate model. In contexts where the machine learning model is not imposed due to external factors, simultaneous optimization of model and hyperparameters is more complex but also more interesting. This is known as CASH, or Combined Algorithms Selection and Hyperparameters optimization. When the models in consideration are limited to neural networks, the model selection process involves automating the design of the neural network architecture (e.g., the number and type of neurons, and the connections between them), and is referred to as Neural Architecture Search (NAS). When HPO is integrated with NAS, the resulting optimization problem is designated as Automated Deep Learning, or AutoDL.

All these different optimization problems have three main elements in common: the search space, the search algorithm, and the performance evaluation.The search space must contain all possible combinations of the elements to optimize.The search algorithm is used to navigate through the search space to find good configurations, and the performance evaluation is used to estimate the performance of the solutions in the search space. In the following section, we will present a formal definition of the AutoML problem and describe how these three elements have been designed for hyperparameter optimization, architecture search, and AutoML systems.We will focus on those elements that we believe are most relevant for this thesis. For a more comprehensive overview, please refer to the book Hutter et al. (2019), which defines the essential elements of AutoML, and the article Baratchi et al. (2024), which outlines the current state-of-the-art in AutoML.

1.2.1 Problem Definition

AutoML is based on several optimization problems that must be formally defined. The objective is to design optimization algorithms that optimize the (hyper-)parameters of machine learning models, which are themselves optimization algorithms. For instance, in linear regression, predictions are a linear combination of features. The objective is to identify the optimal combination, which is achieved by minimizing the mean squared error between the predictions and the actual values. In decision trees, the splits are optimized to reduce impurity measures such as the Gini index. In Support Vector Machines (SVMs), optimization involves identifying the maximum distance between classes. In the field of deep learning, backpropagation is employed to adjust weights and biases, aiming to minimize the loss function. This process enables the capture of intricate relationships present in data. This thesis does not delve into the algorithms that train a given machine learning model. An AutoML algorithm will optimize the maximum size of splits in a regression tree, the maximum depth of the tree, or determine whether a support vector regressor will perform better on a given task, but will not optimize the values of the splits. We therefore distinguish between two concepts: *model training* and *model optimization*.

Task Definition We consider the supervised learning task, which involves learning a function f from a random variable $X \in E$ to a target variable $Y \in F$. The goal is to approximate the true underlying relationship between X and Y, namely the conditional expectation $\mathbb{E}_{(X,Y)\sim p_{X,Y}}(Y|X)$, where $p_{X,Y}$ is the joint distribution of (X, Y). Given a dataset $\mathcal{D}$ containing independent and identically distributed (i.i.d.) realizations of (X, Y), the task is to find a function f that minimizes a loss function over $\mathcal{D}$. This loss function $\ell_{\text{train}}\big(f(X), Y\big)$ measures the discrepancy between the predicted values $f(X)$ and the actual values Y.

Definition 1.2.1 (Model training)**.** *Given a machine learning model f_δ, with $\delta \in \Delta$ its trainable parameters, training the model f_δ according to a training loss function ℓ_{train}, means finding the optimal parameters $\delta^\star$ such that:*

$$\delta^\star \in \underset{\delta\in\Delta}{\operatorname{argmin}}\, \mathbb{E}_{(X,Y)\sim p_{X,Y}}\Big[\, \ell_{\text{train}}\big(f_\delta(X), Y\big)\,\Big]. \tag{1.1}$$

The trained model is denoted $f_{\delta^\star}$.

We denote Ω the search space containing all the models considered for this task (which could be different models for model selection, or the same model with different hyperparameters for hyperparameters optimization). The *model optimization* is done by evaluating the performance of the already trained models $f_{\delta^\star} \in \Omega$ with regards to an objective (loss) function ℓ.

Definition 1.2.2 (Model optimization)**.** *Given a search space Ω,the model optimization problem consists in finding the optimal trained model $f^\star_{\delta^\star} \in \Omega$ which minimizes an objective function ℓ:*

$$f^\star_{\delta^\star} \in \underset{f\in\Omega}{\operatorname{argmin}}\, \mathbb{E}_{(X_i,Y_i)\sim p_{X,Y}}\Big[\, \ell\big(f_{\delta^\star}(X_i), Y_i\big)\,\Big]. \tag{1.2}$$

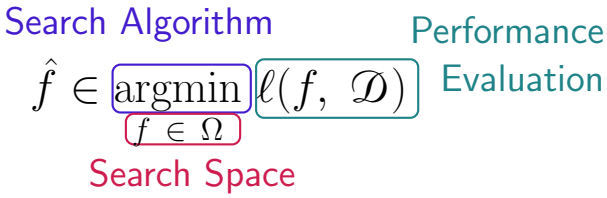
In practice, we do not have access to the expectations. So we minimise the losses ℓ_{train} and ℓ on samples of i.i.d. data. To avoid overfitting, we use two different datasets $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{valid}}$ to perform the two tasks. Thus, the expectation is approximated with:

$$\begin{aligned} \ell \;:\; \Omega &\to \mathbb{R} \\ f \;&\mapsto \ell\big(f_{\hat\delta}, \mathcal{D}_{\text{valid}}\big)\,, \end{aligned}$$

where $\ell\big(f, \mathcal{D}\big)$ is the loss of the model f evaluated on the dataset $\mathcal{D}$ and $\hat\delta$ are the optimal trainable parameters of f calculated when training the model on a training dataset $\mathcal{D}_{\text{train}}$:

$$\hat\delta \in \underset{\delta\in\Delta}{\operatorname{argmin}}\, \ell_{\text{train}}\big(f_\delta, \mathcal{D}_{\text{train}}\big)\,,$$

where the loss used to train the model f on a dataset $\mathcal{D}$ is denoted $\ell_{\text{train}}\big(f, \mathcal{D}\big)$ and may be different from ℓ. As mentioned above, this thesis does not discuss the the algorithms that train a machine learning model but they may depend on some hyperparameters such as the optimizer or the batch size which may be included in Ω. The *model optimization* problem definition here is generic, but can be specified for model selection, hyperparameter optimization, neural architecture search, and CASH problems, depending on the search space. For the CASH problems, two optimization loops can be nested, since certain hyperparameters depend on the type of model. Definition 1.2.2, the search space is Ω, the search algorithm indicates how to find the argmin element $f \in \Omega$, and the performance evaluation is characterised by the loss function ℓ.



The challenge in AutoML will be to define these three elements correctly to conduct an efficient search.

Training a machine learning model is a long and resource-consuming process. Furthermore, this training is stochastic and dependent on the datasets chosen, namely, $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{valid}}$. Optimizing such an expensive, noisy and black-box function is challenging and requires good search algorithms. A search algorithm is typically given a certain budget, $T \in \mathbb{N}^{\star}$, to find the best configuration. This budget can be defined as a specific number of model trainings or a given running time, etc. It is typically much smaller than the number of configurations within the search space. The algorithm must therefore evaluate the most promising solutions within this budget, leveraging past configuration evaluations to do so. This implicitely assumes that there is a distance between configurations, and that two relatively close configurations from the search space will perform similarly. A search algorithm must find the right balance between exploration and exploitation. It must evaluate a large number of distant solutions in the search space and intensify the search for promising solutions and the configurations close to them. This notion of distance can be difficult to define in search spaces that contain different types of hyperparameters (e.g. categorical) or complex objects to represent the models (e.g. tree or graph structures). Given the long training times for machine learning, some algorithms use resource allocation to speed up the process. This involves splitting up the training process (for example, using only a subset of $\mathcal{D}_{\text{train}}$ or train for a small number of epochs for a neural net) to free up resources for promising models, allowing more exploration.

Finally, performance evaluation must identify the best model for a dataset of interest, but which also has a good generalisation capacity on unseen data. To prevent overfitting, the configurations might be trained on train dataset $\mathcal{D}_{\text{train}}$ and evaluate on a

different dataset $\mathcal{D}_{\text{valid}}$ as mentioned above. A final $\mathcal{D}_{\text{test}}$ may assess the final score of the AutoML framework. Techniques such as cross-validation can also be applied, but not in the case of energy time series forecasting as the samples are not i.i.d. For this particular task, the three datasets follow each other in time, the oldest being $\mathcal{D}_{\text{train}}$ and the most recent $\mathcal{D}_{\text{test}}$ as shown Figure 1.4. The hypothesis implied by this setting is that the three datasets share the same distribution $p_{X,Y}(X, Y)$.

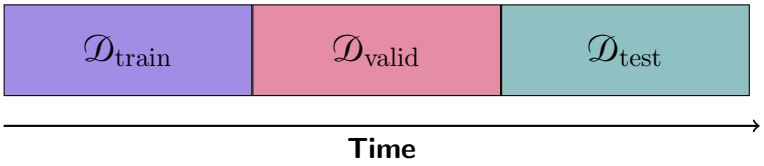


Figure 1.4: Data splits over time.

The definition of the three elements: the search space, the search algorithm and the performance evaluation depends on the task at hand. We will detail examples of such choices for the Hyperparameters optimization, the Neural Architecture Search, and the general AutoML tasks.

1.2.2 Hyperparameters optimization

The Hyperparameter optimization Problem (HPO) is finding the optimal hyperparameters for a given machine learning model. We have already chosen the machine learning model to optimize and are looking to find the best version of that model for a given dataset.

1.2.2.1 Search Space

The hyperparameters of a machine learning model are most commonly encoded using an array of fixed size, with each variable containing the value for a hyperparameter. They can be either numerical or categorical. Numerical hyperparameters, such as the maximum depth of a regression tree or the learning rate of a neural network, are a subset of $\mathbb{N}$ (for integer hyperparameters) or $\mathbb{R}$ (for float hyperparameters). They fall within a range between two extreme values. Categorical hyperparameters are discrete and have no order between the candidate choices. For example, the k-nearest neighbor classifier can take different functions to compute the weights of each point within the neighborhoods. These hyperparameters are generally encoded using numerical encodings, such as one-hot encodings, within the array. Furthermore, there may also exist dependencies within the hyperparameters. Some hyperparameters are only valid if certain other hyperparameters have been chosen. For example, the kernel coefficient gamma of a support vector regression model is only valid for certain kernel types. These are called conditional hyperparameters. Tree-based structures or variable-size arrays are possible choices for search spaces containing conditional hyperparameters. For an overview of mixed and

conditional search spaces, see Gamot (2023). The majority of research on search spaces for HPO focuses on reducing the search space as detailed by Baratchi et al. (2024). This can be achieved through expertise, historical data, search space analysis, and the importance of hyperparameters, as demonstrated by the functional analysis of variance (ANOVA, see Hutter et al. 2014).

1.2.2.2 Search Strategy

Once the search space is defined, several search strategies can be implemented to find the best configurations. The simplest and most straightforward approach would be to exhaustively evaluate all configurations from the search space, referred to as a grid search. This approach is particularly efficient when the search space is small and the hyperparameters are encoded as discrete elements. Continuous variables should be mapped to a set of discrete values. However, search spaces are often too extensive to perform a grid search, as the number of configurations to evaluate grows exponentially with the number of hyperparameters. An alternative approach is to use the random search method introduced by Bergstra and Bengio (2012), where configurations are randomly sampled from the search space. This approach is easy to implement and distribute across multiple devices, and it may already be a strong baseline to beat. However, more sophisticated strategies have been developed to select future configurations for evaluation based on prior evaluations.

Bayesian optimization Bayesian optimization (Frazier, 2018) is a sequential optimization technique commonly used to optimize black-box functions. It is based on two main components: a surrogate model that approximates the unknown objective function and an acquisition function that selects the next element in the search space to be evaluated. The surrogate model is employed to estimate the probability of a score within the search space as a function of the scores of previously evaluated points. The objective is to ensure that the surrogate model can be efficiently minimized. It is updated a posteriori each time a new configuration is evaluated, preserving the information from past evaluations to select future evaluations wisely. In the context of HPO, the surrogate model is employed to approximate the function that maps a set of hyperparameters to the corresponding performance score.

The various Bayesian optimization approaches to HPO that exist in the literature tend to differ in the type of surrogate model and fit function. In terms of surrogate models, Gaussian processes (Garrido-Merchán and Hernández-Lobato, 2020) and Random Forests, used for example by the SMAC package from Hutter et al. (2011) and Lindauer et al. (2022), have been studied in several papers. The advantage of random forests over Gaussian processes is that they can handle different types of hyperparameters, whereas Gaussian processes, without any data transformation, can only handle continuous hyperparameters. Regarding the acquisition function, the expected improve-

ment is commonly used by the community. It consists of two terms representing the expected performance at a given point and the associated uncertainty. The two terms represent the exploitation and exploration trade-offs mentioned above. Other sampling functions can also be used, such as the upper confidence bound (Hoffman and Shahriari, 2014) or the entropy search (Hennig and Schuler, 2012).

Another Bayesian optimization variant frequently used in HPO is the Tree-structured Parzen Estimator (TPE), introduced by Bergstra et al. (2011). TPE replaces Gaussian processes with a non-parametric model that divides the search space into densities of "good" and "bad" configurations based on previously evaluated points. This makes it well suited to handling both continuous and categorical hyperparameters in larger, more complex search spaces. The learning function of TPE aims to select new configurations that maximize the expected improvement, ensuring a balance between exploring unknown regions and exploiting promising configurations.

Bandits Algorithms Continuous bandit algorithms conceptually close to Bayesian optimization, such as GP-UCB and KernelUCB (see, respectively, Srinivas et al. 2010 and Valko et al. 2013), which extend the classical Upper-Confidence Bound (UCB) algorithm, have been massively used for optimization and, eventually, HPO. They rely on a notion of distance between configurations. Discrete bandit approaches without distance have also been introduced, such as Hyperband (Li et al., 2018), which dynamically allocates resources to different configurations by treating the process as a multi-armed bandit problem. Hyperband operates through a series of iterations, during which configurations receive progressively more resources (e.g., training time or data) based on their early performance. This approach enables the rapid discarding of configurations with poor performance, thereby favoring those with potential. This approach is particularly well-suited for large search spaces, reducing the overall computational cost by avoiding exhaustive evaluation of all possible configurations. As a result, Hyperband is a highly effective solution for deep neural networks and other resource-intensive models.

Reinforcement Learning Bandits can be regarded as a specific type of Reinforcement Learning (RL) algorithms. Other RL-based algorithms (Jomaa et al., 2019) have also been applied to HPO. In RL-based HPO, an agent adjusts hyperparameters iteratively to maximize a reward signal, typically defined by the model's performance on validation data. This approach enables the adaptive, stepwise refinement of hyperparameters and can be particularly useful in conditional search spaces.

Evolutionary Algorithms Another popular approach is evolutionary algorithms (EA). Inspired by natural selection, EA works by evolving a population of hyperparameters configurations over successive generations (Young et al., 2015). EA typically includes operations such as selection, mutation, and crossover to modify the configuration, allowing the exploration of complex search spaces in a non-deterministic manner. Mutation

can be thought of as a function that selects the new value within the neighborhood of the current value. Crossover combines two or more configurations to transfer the good genes to the offspring. The two operators represent respectively exploitation and exploration respectively.

Gradient-based optimization Finally, gradient-based optimization methods offer an efficient approach to HPO for differentiable models and continuous hyperparameters, particularly in deep learning settings. Methods such as bi-level optimization leverage gradients of the model's validation loss, $automlloss$, with respect to the hyperparameters to directly update them, leading to substantial reduction in search time (Franceschi et al., 2018). While gradient-based methods are highly efficient, they are generally limited to continuous and differentiable hyperparameters, which poses challenges when applied to categorical or discrete settings.

For a more comprehensive and recent review of HPO search spaces, search algorithms and systems, see the work of Baratchi et al. (2024).

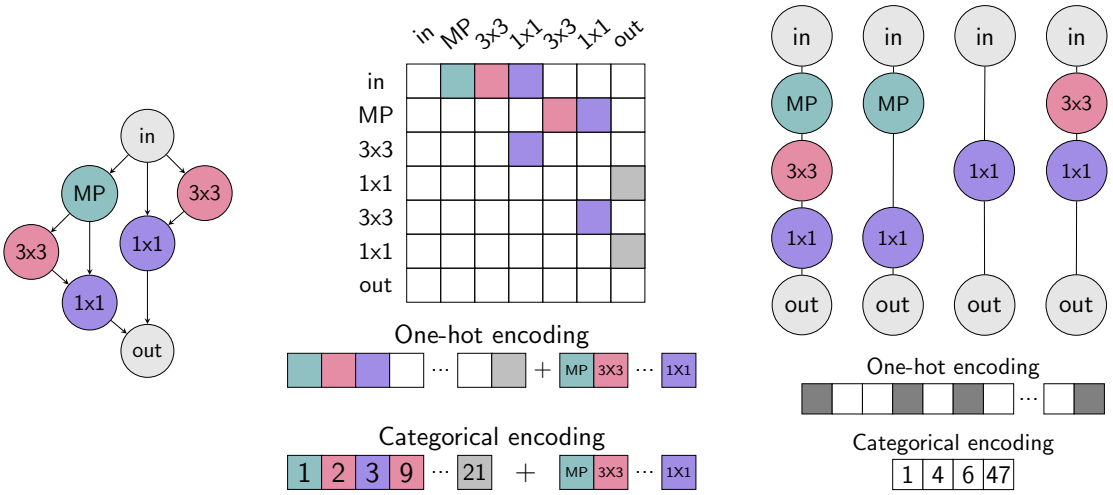
1.2.3 Neural Architecture Search

Among the various model selection approaches, one sub-field has garnered significant attention: Neural Architecture Search (NAS). Rather than selecting the optimal match among a range of machine learning algorithms, NAS focuses exclusively on deep neural networks, aiming to optimize their architecture. The variables that a NAS algorithm should optimize can be divided into two categories: training hyperparameters and architectural parameters. Training hyperparameters, such as the learning rate or optimizer type, influence the training procedure (i.e. Definition 1.2.1). Architectural parameters, such as the type or number of layers, define the network architecture. While NAS is conceptually similar to HPO, representing the architecture of a neural network is more complex. Depending on the number of design choices contained in the search space, the connections between layers and the hyperparameters of these layers can be modeled, making direct array representations quickly insufficient.

1.2.3.1 Search Space

In Neural Architecture Search (NAS) frameworks, the search space should contain all possible neural architectures suitable for a given problem, which can become large and complex.

Macro-level search space One of the earliest approaches was NEAT (for NeuroEvolution of Augmenting Topologies, see Stanley and Miikkulainen 2002). NEAT represents the architecture at the neuron level and demonstrates the potential to evolve both network weights and topologies for complex tasks. Over a decade later, the focus shifted to gradient descent for network weights (a topic not covered in this thesis), and more



(a) Neural architecture.

(b) Adjacency matrix representation of the architecture showing two encodings

(c) Path-based representation, showing two encodings

Figure 1.5: Adjacency matrix and path-based representations for macro-level search spaces. The figure has been reproduced from the paper White et al. (2020).

constrained macro-level search spaces were introduced. These macro-level search spaces explore entire layers and network-wide parameters, including the number and type of layers, connections, and filter sizes (but not the trainable weights).

Macro-level search spaces were utilized by Baker et al. (2022), who introduced MetaQNN, a reinforcement learning-based method for automatically generating high-performance Convolutional Neural Network (CNN) architectures. The architecture is a sequence of a maximum number of layers. Each layer is associated with several candidate operations, including convolution, pooling, and feed-forward. Each operation is associated with a specific set of hyperparameters, including termination hyperparameters that determine the final layer of the architecture. The neural networks are represented as graphs, with operations defined on the nodes.

Common representations for those graphs, as detailed by White et al. (2020), are through the adjacency matrix on one side, which represents the architecture as a list of edges and operations, or path-based approaches, which represent the architecture as a set of paths from the input to the output. One-hot and categorical encodings are then used to pass the configuration to various search algorithms. Figure 1.5 illustrates both representations and encodings. These approaches impose significant constraints on the representable network diversity. The number of nodes and the number of candidate operations are restricted. Additionally, the hyperparameters of the latter cannot be optimized.

Weights sharing To accelerate the search strategy, weight sharing, also known as one-shot architecture search, was implemented. The search space is contained within a single large network capable of emulating any architecture considered. Training is conducted on this extensive architecture, and at the completion of the training process, the optimal sub-architecture is retained, as illustrated in Figure 1.6. SMASH, a one-shot architecture search method introduced by Brock et al. (2018), involves gradually pruning a super-network by setting some branches to zero to obtain the final model. This representation can be memory-expensive, as it requires the simultaneous training of all the candidate choices. The macro-level approach offers greater flexibility in exploring novel architectures; however, it is less efficient computationally. Researchers have explored more constrained and efficient alternatives due to the vastness of macro-level search spaces.

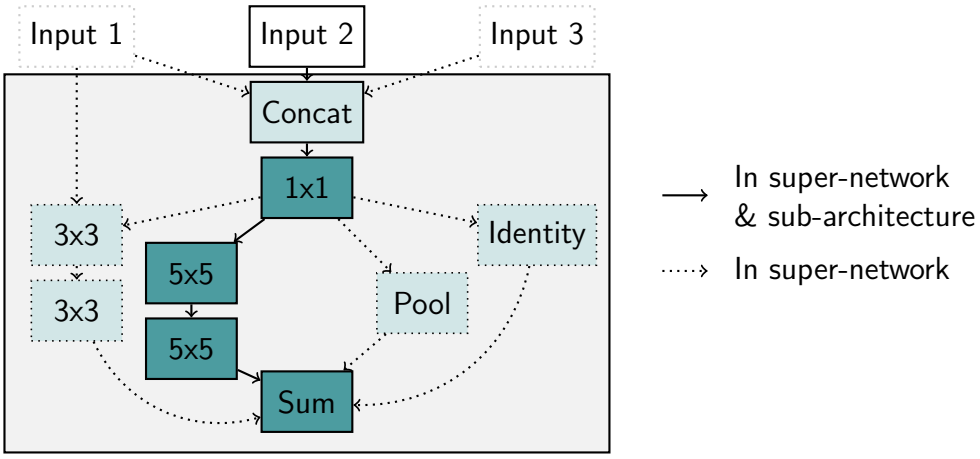


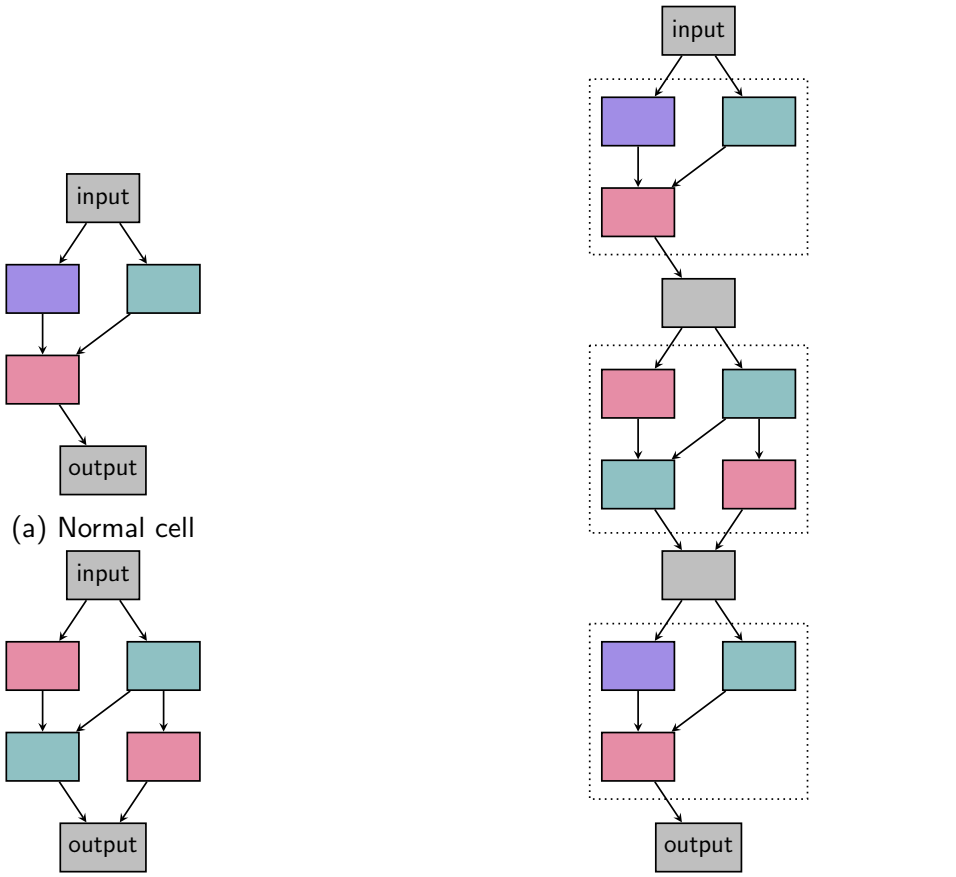
Figure 1.6: An example of a one-shot cell that receives three inputs, concatenates them, applies a 1×1 convolution operation, followed by multiple other operations, and finally sums the result together. A sub-architecture, denoted by solid edges, can be evaluated by disabling inputs and operations by zeroing them out.

Cell-based search space The period between 2017 and 2019 saw a number of significant contributions to NAS in computer vision tasks, including works from Zoph and Le (2016), Zoph et al. (2018), Real et al. (2019), Liu et al. (2018d), or Elsken et al. (2019). Successful computer vision architectures, such as ResNet (He et al., 2016), are characterised by repetitive patterns of convolutional and pooling layers. This has led to the introduction and exploration of cell-based search spaces.

In this paradigm, a neural network consists of one or more cells that are stacked repetitively. Each cell is represented as a small Directed Acyclic Graph (DAG), as shown in Figures 1.7a and 1.7b. The nodes in the graph correspond to operations selected from a predefined set. These operations generally include 3×3 and 5×5 convolutions, pooling, and occasionally more advanced operations such as dilated convolution or separable convolution. The DAG edges denote the connections between these operations. Typically,

the input connections for each node are limited to two, a design choice that simplifies the architecture while ensuring that the graph remains manageable. Additionally, the number of nodes within a cell is often fixed, allowing the entire cell to be encoded as a fixed-length vector, which facilitates the application of various optimization algorithms (Zoph et al., 2018).

Hierarchical search space Hierarchical search spaces (Liu et al., 2018c) build upon the cell-based concept by organizing cells within a higher-level structure. This approach enables architectures with multiple levels or modules, offering a more sophisticated framework for complex tasks. In these spaces, architectures can be constructed by stacking different types of cells (e.g., reduction cells for downsampling and normal cells for feature extraction) in a multi-level configuration (see Figure 1.7), resulting in highly expressive models. Hierarchical search spaces enable NAS algorithms to adapt network depth, width, and cell composition dynamically, potentially resulting in more complex architectures suitable for a broader range of tasks (Liu et al., 2018a).



(a) Normal cell

(b) Reduction cell

(c) Architecture built by stacking the cells.

Figure 1.7: Example of an architecture found with a hierarchical search space based on two cells: normal and reduction cells.

Morphing-based search space Recently, morphing-based search spaces, such as ProxylessNAS by Cai et al. (2018b), have been proposed as an efficient alternative, particularly useful when starting from a strong, manually designed base model. In this approach, the search algorithm starts with an existing architecture and iteratively "morphs" it by adding or modifying layers, connections, or filter sizes. This approach, characterized by its constrained yet adaptive nature, has been shown to reduce the computational burden compared to a full search. Rather than generating entirely new architectures from scratch, the search focuses on incremental modifications.

Embeddings To leverage search algorithms that operate in continuous search spaces, efforts have been made to develop embedding-based search spaces. These spaces represent architectures as embeddings in a continuous latent space. These embeddings are typically learned using techniques such as variational autoencoders (Chatzianastasis et al., 2021) or graph neural networks (White et al., 2020). Embedding-based search spaces have proven effective in handling diverse architectures and supporting fine-grained optimization. However, they may face challenges in capturing complex architectures.

Context-free grammars Recognising that NAS has primarily been applied to over-engineered, restrictive search spaces (e.g. cell-based) that have not given rise to truly novel architectural patterns (White et al., 2023), recent advances in NAS have introduced the use of context-Free Grammars (CFGs) to construct hierarchical search spaces. This approach, as described in Schrodi et al. (2024) and illustrated Figure 1.8, exploits the recursive nature of CFGs to generate highly expressive and comprehensive search spaces that can encompass a wide range of architectural patterns. By using CFGs, researchers can define a unifying framework that allows for efficient exploration of both micro and macro architectural levels, promotes regularity, and allows for the incorporation of user-defined constraints. This method has shown promise in improving the efficiency and effectiveness of NAS by allowing search across complete architectures rather than limited aspects, paving the way for the discovery of novel and high-performing neural network designs.

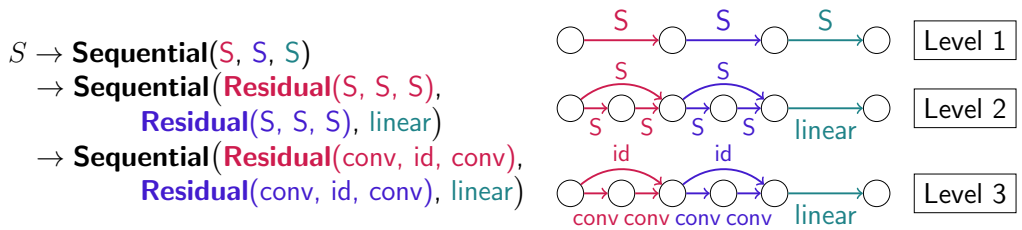


Figure 1.8: Context-free grammar based search space from Schrodi et al. (2024). On the right is the graph representation of the function composition of the neural architecture from the equation on the left.

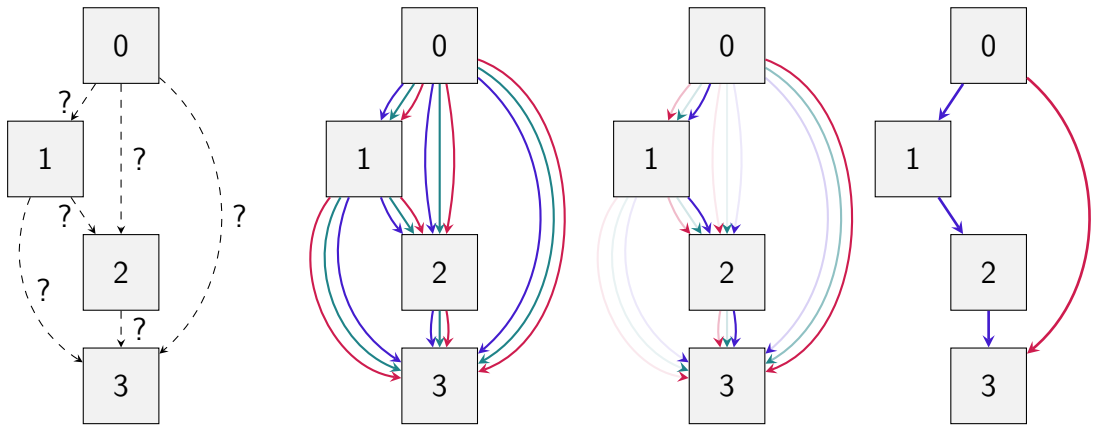
NAS's diverse search spaces enable it to address a range of challenges, striking a balance between flexibility, creativity, and efficiency for specific tasks. Cell-based and hierarchical spaces are particularly effective for achieving high performance at manageable search costs, while macro-, morphing- and embedding-based spaces provide flexibility to search for complex or customized architectures. The selection of search space is therefore a critical factor in NAS, directly impacting both the computational feasibility and the effectiveness of the search.

1.2.3.2 Search Strategy

The primary challenge in NAS search strategies is the complex and often discrete nature of the search space. This includes connectivity patterns and operations that do not fit into a continuous vector like traditional hyperparameters. This discrete nature of the space poses significant challenges to the application of search strategies. Researchers have developed several NAS methods, which are usually applicable for a specific type of search space.

Reinforcement-Learning RL-based NAS, as seen in Zoph and Le (2016), was among the first approaches, where an RL controller sequentially generated architectures by predicting discrete decisions, like selecting layer types or connections, and received rewards based on the model's accuracy. Despite its flexibility, this approach is computationally intensive, as each new architecture requires training from scratch, making it costly in terms of time and resources. To overcome such inefficiencies, Efficient NAS (ENAS) introduced by Pham et al. (2018) exploits the idea of supernetwork by searching for an optimal subgraph within a large computational graph representing all design choices. These approaches avoid having to train each candidate model from scratch. However, weight sharing can sometimes lead to suboptimal final architectures, as the shared weights may not transfer perfectly to individually trained architectures.

Differentiable search The DARTS (Differentiable Architecture Search) method introduced by Liu et al. (2018d) made another significant step by transforming NAS into a differentiable problem. DARTS employs a supernetwork, assigning a probability to each candidate operation of being selected in the final architecture. These probabilities are optimized by gradient descent with respect to the validation loss. The optimization process is illustrated Figure 1.9 This approach significantly reduces search time, making NAS feasible on single GPU setups. However, DARTS relies on approximating the architecture selection in a differentiable way, which does not directly translate to the discrete selections required in actual neural networks, sometimes resulting in architectures that are less robust.



(a) Step 1. (b) Step 2. (c) Step 3. (d) Step 4.

Figure 1.9: Differentiable Architecture Search (DARTS) optimization scheme. Step 1 is depicted Figure 1.9a: operations on the edges are initially unknown. Step 2 is depicted Figure 1.9b: continuous relaxation of the search space by placing candidate operations on each edge. Step 3 is depicted Figure 1.9c: joint optimization of the probabilities and the model weights by solving a bilevel optimization problem. Finally, step 4 is depicted Figure 1.9d: inducing the final architecture from the learned probabilities.

Evolutionary Algorithms Evolutionary algorithms (EA) provide an alternative solution by leveraging genetic programming techniques to explore the search space. AmoebaNet, introduced by Real et al. (2019), incorporates mutations, crossover, and selection to systematically evolve a population of architectures within a hierarchical cell-based search space. This approach is highly flexible and adapts well to graph representations; however, it often requires substantial computational resources.

Bayesian optimization Bayesian optimization (BO) with Gaussian processes is only applicable to continuous variables. To apply BO on NAS, one strategy is to enforce a continuous search space using embedding methods, as it has been done by Chatzianastasis et al. (2021). Alternatively, new distance metrics have been proposed to measure the similarity between graph-based NAS encoding configurations, such as the network morphology metric from AutoKeras (see Jin et al. 2019). When employing tree-based surrogate models, continuous representations are no longer necessary. For instance, Mendoza et al. (2016) employed the Bayesian optimization (BO) with the random forest surrogate model from SMAC.

Together, these approaches reflect the diversity and complexity of NAS search algorithms. The discrete nature of the search space necessitates innovative methods beyond traditional optimization techniques, and each strategy balances trade-offs between efficiency, search space fidelity, and computational demands.

1.2.4 AutoML systems

Based on the search spaces and search algorithms mentioned above, AutoML systems have been developed by both academia and industry to address model selection, HPO, NAS, or directly the CASH problem. Some of these systems have been implemented as no-code interfaces, allowing users with limited or no experience in machine learning to access cutting-edge solutions directly.

Early developments in the 2010s led to the emergence of frameworks such as Auto-WEKA (see Thornton et al. 2013) in 2013. The framework is based on the WEKA library and uses Bayesian optimization for automated model selection and hyperparameter optimization. Auto-WEKA relies on a large search space of algorithms and hyperparameters, making it versatile but resource intensive. Following Auto-WEKA, TPOT (Olson and Moore, 2016) introduced genetic programming for optimizing scikit-learn (Kramer and Kramer, 2016) machine learning pipelines, potentially creating complex pipelines, and Auto-sklearn (Feurer et al., 2015), developed in 2015, integrates meta-learning and ensemble construction.

Commercial solutions began to emerge with companies like DataRobot[3] (founded in 2012), which automates the end-to-end machine learning process with a no-code interface. Optuna (Akiba et al., 2019), developed by Preferred Networks and introduced in 2017, is a hyperparameter optimization framework that uses cutting-edge algorithms for efficient search and pruning that can be applied across various machine learning frameworks. Google Cloud AutoML (Bisong et al., 2019), introduced in 2018, provides a suite of tools for building custom machine learning models with minimal coding and support for various data types.

AutoML systems dedicated to automated deep learning or NAS started with AutoKeras (Jin et al., 2019), introduced in 2018 and based on the Keras library, which focuses on neural architecture search using Bayesian optimization. The NAS framework for *Pytorch* called Auto-PyTorch was released in 2020 and combines neural architecture search with hyperparameter optimization, supporting tabular data (Zimmer et al., 2021) and time series forecasting (Deng et al., 2022).

Recently, AutoML libraries based on a variety of models have been developed. For example, AutoGluon (see Erickson et al. 2020 for the tabular version and Shchur et al. 2023 for the time series one), developed by Amazon in 2020, supports multimodal data (see Tang et al. 2024) with stacking and ensembling, providing flexibility but facing interpretability challenges due to model complexity and the black-box nature of its algorithms. Its competitor, FLAML (Wang et al., 2021), from Microsoft Research in 2021, prioritizes efficiency with lightweight search strategies, making it suitable for resource-constrained environments.

Finally, basic models have also been studied, such as TabPFN (Hollmann et al., 2023), published in 2023, which uses a transformer-based approach for small tabular data sets,

[3] https://www.datarobot.com/

providing fast performance but limited to smaller data sizes. A study by Gijsbers et al. (2024) compared some of these systems on various regression and classification tasks and found that AutoGluon consistently had the highest average rank in their benchmark. Many of these systems, such as Auto-sklearn, TPOT, and Optuna, are widely used in both academic research and industrial applications due to their robust performance and flexibility.

1.3 AutoML for energy forecasting

Having established the value of AutoML for the electricity sector and reviewed the major developments in this area, we now review how these two areas have been combined to date. A small number of studies have been conducted using HPO, AutoDL, and AutoML methods for load and production forecasting tasks.

1.3.1 AutoML systems

A number of the setups mentioned in Section 1.1.2 for production and load forecasting can be represented as a regression problem on tabular data. The various explanatory variables can be put into a one-dimensional vector $X_t \in \mathbb{R}^F$, where $F \in \mathbb{N}^\star$, which can be mapped into the target variable $Y_t \in \mathbb{R}$ via a regressor model. A number of papers have used this representation to successfully apply AutoML frameworks to load and wind power forecasting.

Therefore, Wang et al. (2019b) applied AutoML to the load forecasting problem for the first time. Specifically, they used two systems, namely Auto-sklearn (Feurer et al., 2015) and TPOT (Olson and Moore, 2016), and showed that AutoML can reduce the effort to build performing predictive models for the load forecasting task. More recently, Meng et al. (2022) compared the performance of tabular AutoGluon (Erickson et al., 2020), tabular AutoPytorch (Zimmer et al., 2021), and Auto-Keras (Jin et al., 2019) on a regional short-term load forecast, and found that AutoGluon outperformed the other two AutoML frameworks on this dataset, and Koutantos et al. (2024) used FlaML (Wang et al., 2021), AutoML by H2O. AI (LeDell and Poirier, 2020) and AutoTS (Wang et al., 2024) to forecast the national Greek load. Kovalevsky et al. (2022) estimated the coefficients of a wind turbine power using Auto-WEKA (Thornton et al., 2013), Hyperopt-sklearn (Komer et al., 2014) and Auto-Sklearn. They showed that forecasts using the power curve estimated by the AutoML frameworks were better than those obtained with theoretical power predictions based on physical approximations. Tu et al. (2022) performed hyperparameter optimization on deep learning models for wind power forecasting using Optuna (Akiba et al., 2019) or SMAC3 (Lindauer et al., 2022) and argued that the AutoML frameworks did not perform this task well enough to be preferable to manually designed models.

This work demonstrates the potential of HPO and AutoML systems for load and production forecasting. The representation of forecasting problems as regression problems allows the state-of-the-art AutoML frameworks from the tabular data community to be directly applied in a fairly powerful way. However, the articles that apply these frameworks to load and wind power forecasting tasks generally limit themselves to comparing AutoML methods with each other or with fairly simple baselines, rather than with state-of-the-art forecasting models. One might speculate that their performance might be somewhat lower because regressors on tabular data assume that the rows are independent and identically distributed, which is not the case between different time steps of load or production curves.

1.3.2 Automated Deep Learning

Most automated deep learning research focuses on optimizing convolutional and recurrent network architectures applied to image or text data. These are difficult to apply to load and production data. However, some work has been done on optimizing neural networks for energy time series forecasting. The general approach is to use HPO techniques and include some architecture-related hyperparameters in the optimization, such as the number of layers.

Therefore, Firmin et al. (2021) used various metaheuristics and ensembling techniques to optimize the hyperparameters of a deep learning model to forecast French load consumption. Architectural hyperparameters such as the number of layers and whether or not to add a dropout layer were among the elements to be optimized. Pujari et al. (2023) implemented an evolutionary-based HPO framework to optimize a deep convolutional LSTM architecture for wind power forecasting. Jalali et al. (2021) used a reinforcement learning algorithm to optimize a stacked architecture of nonlinear autoregressive neural network, wavelet neural network, and LSTM to model the wind characteristic data to predict the wind power. Finally, Yang et al. (2024) extended the Efficient Architecture Search (EAS) framework of Cai et al. (2018a), originally created to optimize convolutional architectures, and added recurrent neural networks to the available layers. They applied their work to load and wind power forecasting.

The search spaces in these works are mostly arrays of fixed size containing the various hyperparameters to be optimized. These approaches reveal a clear gap in the literature: there is a need for effective ways to encode architectures for tasks broader than image and text processing. While these architectures consist of repetitive patterns of the same layer sequences, other tasks require models that combine different types of layers, with hyperparameters that affect performance almost as much as the architecture itself. Such models cannot be represented within existing search spaces, nor are current search algorithms versatile and efficient enough to handle such a large number of design choices. While this observation is made here in the context of energy forecasting, it has also been noted by Tu et al. (2022) for applications related to climate change that

deal with physically constrained problems or spatio-temporal data, and by Baratchi et al. (2024), who noted the lack of AutoDL search spaces designed for more diverse types of data, whether unstructured, temporal, or spatio-temporal.

1.3.3 Open problems

The research teams at EDF have developed highly sophisticated models for forecasting load and wind power production. However, as the number of signals to be forecasted continues to grow, there is an increasing need for automating the creation of efficient forecasting models. AutoML is an ideal community for addressing this challenge. However, the tools developed to date are limited to regressors for tabular data, time series forecasting models, and neural networks for images and text data. These approaches are insufficient for meeting the needs of energy forecasting. They struggle to effectively handle time series data, which often includes dependencies on numerous explanatory variables in various formats (e.g., weather grids, calendar indicators, graphs, text). Additionally, these tools are limited in their ability to incorporate advanced energy forecasting techniques, such as generalized additive models (GAMs) or CNN-LSTM neural networks, into their search space. This thesis aims to address these identified issues, thereby bridging the gap between the work of the AutoML community in automating the creation of high-performance machine learning models and the specific model requirements of electricity forecasters.The research directions we consider most promising in this context fall into three main categories.

1. **Search Spaces.** The search spaces from the literature do not encompass some of the state-of-the-art models for production and load forecasting such as GAMs or various neural networks.

2. **Search Algorithms.** A search space encompassing a wide range of models is not only very large, but also made up of a variety of objects. This complicates the search process. The development of new algorithms that are both efficient and versatile is essential. These algorithms should be capable of handling a variety of models and their diverse representations.

3. **Creating a software for electricity forecasters.** AutoML softwares are designed with one primary focus: their users. Electrical forecasting applications are diverse. It is crucial to provide a tool that is both efficient and easy to use, while also allowing users to customize it to their specific needs. The search space, search algorithm, and performance evaluation must be sufficiently modular to adapt to various and complex tasks. Additionally, users should have the flexibility to integrate their own expertise, either to accelerate the search process or to enhance the performance of the system.

1.4 Contributions

This thesis is an initial step in addressing these challenges by optimizing deep neural networks for load and production forecasting. There is a clear opportunity here, as while neural networks are already used in many areas such as image and text processing, they are not yet widely used for time series forecasting or regression – including in the energy sector. AutoDL has centered on enhancing architectures in domains where deep learning has already demonstrated exceptional performance. The next step is to assess its potential in areas where it has not yet demonstrated its capabilities.

In this thesis, we introduce DRAGON, for DiRected Acyclic Graph optimizatioN, an open-source Python package offering multiple tools to optimize deep neural networks for various applications. DRAGON is a toolbox that can be used to create AutoML frameworks for specific applications. Our contributions are separated into two Parts. In Part II we present the algorithmic approach developed for AutoDL and introduce DRAGON, the software we have created to support this work. In Part III we detail how DRAGON can be used for energy sustainability with applications to load and wind power forecasting.

An algorithmic framework for the optimization of deep neural networks architecture and hyperparameters. While Neural Architecture Search has seen significant study in recent years, the proposed works generally utilize rigid, manually-designed search spaces. These optimization frameworks are effective for image and text-based tasks, but extending them to other areas is challenging.

In this work, we propose a new method for encoding Deep Neural Networks (DNNs) that allows for more flexible search spaces than those existing in the literature. As in many of the search spaces in the literature mentioned above, whether at the micro or macro level, DNNs are encoded by Directed Acyclic Graphs (DAGs), where the nodes are operations. However, in most cases, these operations belong to a fixed, reduced list, and the hyperparameters cannot be optimized. In contrast, our representation allows for the inclusion of any *PyTorch* operation selected by the user within the search space. It is also possible to choose the hyperparameters to optimize and in which range of values. This approach enables the creation of novel DNN structures by combining different operations (e.g., convolutional, recurrent, attention-based, etc.). Finally, in addition to the operations, the structure of the DAG is fully optimized. In contrast to the search spaces documented in the literature, there are no restrictions on the number of edges that can arrive at a node.

We are proposing specific neighborhoods and evolution search operators to modify and optimize those graphs. These search operators are compatible with any metaheuristic that can manage mixed search spaces. We demonstrate the value of such a search space and search operators on a time series forecasting benchmark implementing an evolutionary algorithm. The results demonstrate the superiority of our AutoDL approach,

surpassing state-of-the-art handcrafted models and AutoML techniques for time series forecasting on multiple datasets, while also enabling the creation of innovative neural architectures.

> Keisler, J., Talbi, E.-G., Claudel, S. Cabriel, G. **An algorithmic framework for the optimization of deep neural networks architectures and hyperparameters.** *Journal of Machine Learning Research (JMLR)*, 2024.

Mutant-UCB: towards flexible and efficient search algorithms. It is not possible to manipulate the DAGs with complex nodes used in our search space for most efficient search algorithms, such as BO or gradient-based methods. Given the design of the neighborhoods, the evolutionary algorithm was the obvious choice to navigate this space. However, it should be noted that the experiments required a significant amount of time, in comparison to other AutoDL frameworks. To address this challenge, we developed a novel approach, dubbed Mutant-UCB, to enhance the efficiency of the evolutionary algorithm. This algorithm integrates a bandit-approach with evolutionary operators. This approach maintains the flexibility of the evolutionary algorithm while enhancing resource allocation and selection processes. To illustrate the versatility of Mutant-UCB, the algorithm is presented in the generic context of model selection, with experiments conducted within our search space to optimize deep neural networks (DNNs) for image classification. The efficacy of Mutant-UCB is demonstrated by its superior performance in comparison to a random search, the previously utilized evolutionary algorithm, and the bandit-based algorithm Hyperband, as evidenced by experiments conducted on three open-source datasets.

> Brégère, M., Keisler, J. **A Bandit Approach with Evolutionary Operators for Model Selection: Application to Neural Architecture optimization for Image Classification** *International Conference on Knowledge Discovery and Data Mining (ACM SIGKDD), International Workshop on Resource-Efficient Learning for Knowledge Discovery*, 2024.

Packaging and tutorial for DRAGON. The DAG-based search space and the various search algorithms (including the evolutionary algorithm and Mutant-UCB) that might be used with it have been implemented as an open-source package called DRAGON[4]. The package offers different levels of customization so that DRAGON can be applied to more or less complex tasks, and provides a distributed version of the search algorithms on High Performance Computing (HPC) hardware. The DRAGON package is an effective toolbox for developing AutoDL tools for a wide range of DNN applications. We have

[4]The package documentation is available online: https://dragon-tutorial.readthedocs.io

successfully used it to build the AutoDL frameworks EnergyDragon and WindDragon, which are specifically designed for load and wind power forecasting tasks.

EnergyDragon: automated deep learning for load forecasting. We developed EnergyDragon, an automated deep learning framework, using DRAGON to address the load forecasting task. The forecasting setup is a pure regression one, with past load data excluded from the input features. EnergyDragon automatically selects the features fed to the DNNs during the DNN training in an innovative way and optimizes the architecture and the hyperparameters of the models. We demonstrate that EnergyDragon can identify novel neural networks that surpass existing load forecasting methods and other AutoDL approaches on both French and Norwegian load signals.

Keisler, J., Claudel, S., Cabriel, G., Brégère, M. **Automated load forecasting.** *International Conference on Automated Machine Learning (AutoML)*, 2024.

Beyond EnergyDragon: automated spatio-temporal weather modeling for load forecasting. This initial AutoDL approach for load forecasting demonstrated significant promise; however, the forecasting pipeline was not yet fully automated in the initial version of EnergyDragon. The input weather data was already processed using domain knowledge. Weather forecasts from various weather stations were aggregated to create a unique and fixed national indicator. However, it is important to note that the spatio-temporal dependencies of load to weather may change over time or from one signal to another. It is evident that spatial variations do not have a uniform impact, given that consumers are not evenly distributed across the territory. In contrast, temporal variations can have delayed effects on load due to the thermal inertia of buildings. We had the conviction that DNNs were capable of automatically extracting relevant information from raw observations from different weather stations. Consequently, we have integrated an automated representation of spatio-temporal weather modeling for load forecasting into EnergyDragon. The results demonstrate the competitiveness of this approach and its superior performance in comparison to the domain-based weather representation that has been employed in the framework to date.

Keisler, J., Brégère, M. **Automated spatio-temporal modeling for load forecasting.** *International Ruhr Energy Conference (INREC), Best paper award*, 2024.

WindDragon: automated deep learning for regional wind power forecasting from wind speed maps. We addressed a secondary use case with DRAGON: regional

wind power forecasting, leading to the development of WindDragon. We successfully performed short-term (1 to 6 hour horizon) wind power forecasting for the twelve French regions by representing the problem as a regression task over a Numerical Weather Predictions (NWP) wind speed map. Our findings demonstrated the superior performance of WindDragon over forecasts from the French TSO, as well as a handcrafted neural network and other models developed by the regional wind power forecasting community for each region. The use of Mutant-UCB as a search algorithm was instrumental in rapidly identifying effective models for each region. A preliminary version appeared in:

> Keisler, J., Le Naour, E. **WindDragon: Enhancing Wind Power Forecasting with Automated Deep Learning.** *International Conference on Learning Representations (ICLR), Tackling Climate Change with Machine Learning Workshop*, 2024.

The manuscript incorporates the extended version:

> Keisler, J., Le Naour, E. **WindDragon: Automated Deep Learning for regional wind power forecasting.** *Submitted*, 2024.

1.5 Structure of the thesis

The structure of the manuscript and the links between the chapters are illustrated Figure 1.10. The chapters content are detailed below. Part II introduces the algorithmic objects created for Automated Deep Learning in this thesis as well as the package implementation of these objects.

- Chapter 2 presents a new encoding strategy for Directed Acyclic Graph-based Deep Neural Networks (DNNs). This encoding allows the creation of a large variety of DNNs, making it applicable to a wide range of data types and tasks. To navigate efficiently in such a search space, search operators are defined to modify the representations. They are used to implement an evolutionary algorithm. The effectiveness of the representation and search algorithm is demonstrated on a time series forecasting benchmark.

- Chapter 3 introduces Mutant-UCB, a new model selection algorithm that combines the bandit approach and evolutionary operators. This search algorithm is designed to overcome some of the shortcomings of the evolutionary algorithm. By using resource allocation and a bandit-based selection strategy, it becomes more efficient than the evolutionary algorithm of Chapter 2. We prove this by comparing the two algorithms on our search space using three datasets for image classification.

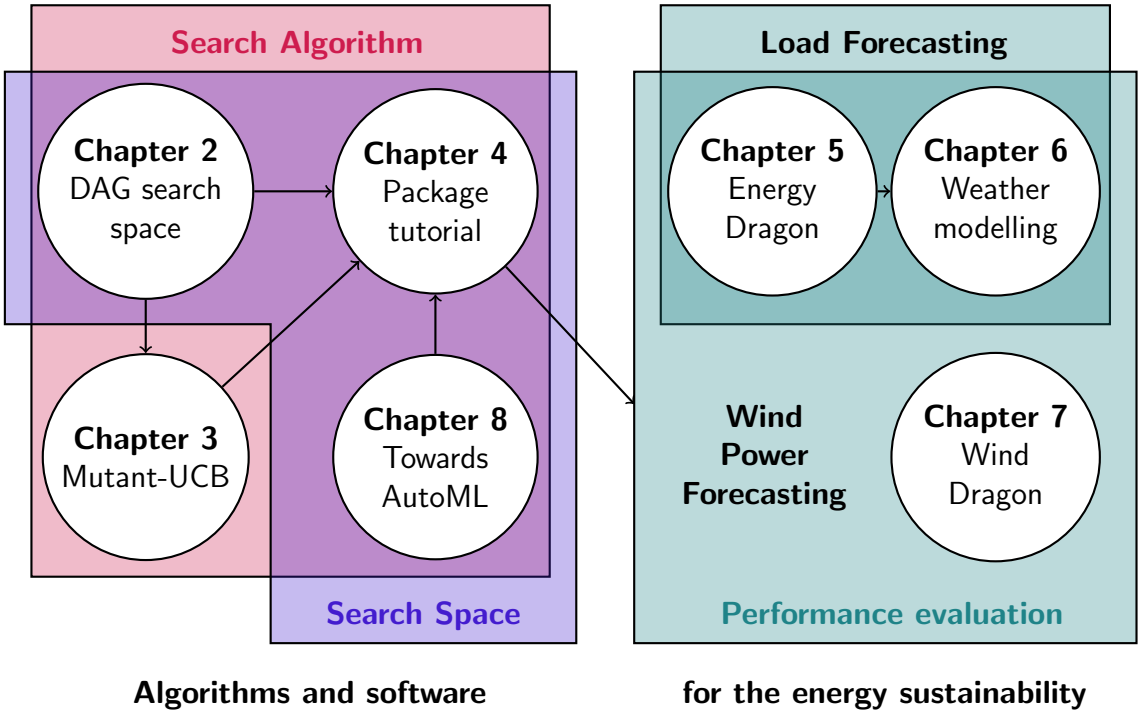


Figure 1.10: Contents and organization of the thesis manuscript

- Chapter 4 provides a tutorial on the DRAGON package, which is the Python implementation of the DAG search space and the various search algorithms. The tutorial lists the package objects, gives examples of how to use them, and details how the software can be distributed on HPC clusters.

Part III presents how DRAGON has been successfully applied to energy forecasting tasks.

- Chapter 5 introduces EnergyDragon, an automated deep learning framework for load forecasting. EnergyDragon automates the feature selection and optimizes the architecture and hyperparameters to forecast load consumption. The experiments on the French and Norwegian national load signals demonstrate the effectiveness of EnergyDragon.

- Chapter 6 iterates over Chapter 5 to further automate the deep learning pipeline for load forecasting. This chapter details the integration of weather data processing automation into EnergyDragon.

- Chapter 7 introduces WindDragon, an automated deep learning framework for wind power forecasting. WindDragon generates DNNs to regress from numerical weather prediction grids to the wind power signal for the twelve French regions.

Finally, Part IV concludes the manuscript.

- Chapter 8 presents a global summary of the thesis and identifies some perspectives to develop DRAGON, currently focused on neural networks, into a more generic AutoML tool, encompassing other models and other types of learning.

Part II

Automated Deep Learning: algorithms and software

Chapter 2

An algorithmic framework for the optimization of deep neural networks architectures and hyperparameters

In this chapter, we introduce the algorithmic framework for the optimization of deep neural networks architectures and hyperparameters that serves as a basis for the package DRAGON. We first detail how the representations of Deep Neural Networks (DNNs) within Automated Deep Learning systems are limited to apply them to tasks other than image or text processing. We take the example of time series forecasting to support our argument. We then propose a new way of encoding DNNs that allows for more flexible search spaces than those existing in the literature. We used Directed Acyclic Graphs (DAGs), where the nodes are DNNs layers. They can be any *PyTorch* operation parameterized by any hyperparameters that the user wants to include in the search space. We propose specific neighborhoods and evolution search operators to modify and optimize those graphs. These search operators can be used with any metaheuristic capable of handling mixed search spaces. We demonstrate the relevance of such a search space and search operators on a time series forecasting benchmark implementing an evolutionary algorithm. The results show that our framework approach outperforms state-of-the-art handcrafted models and AutoML techniques for time series forecasting on numerous datasets while being able to create innovative neural architectures.

Keisler, J., Talbi, E-G., Claudel, S., and Cabriel, G. **An algorithmic framework for the optimization of deep neural networks architectures and hyperparameters.** *Journal of Machine Learning Research (JMLR)* 2024.

Contents

2.1 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, automated 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., 2019a; 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 neighborhoods and variation operators for DAGs. With these operators, any metaheuristic designed for a mixed and variable-size search space can be applied. In this chapter, 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 chapter is organized as follows: we review Section 2.2, the literature on deep learning models for time series forecasting and AutoML. Section 2.3 defines our search space. Section 2.4 presents our neighborhoods and variation operators within the evolutionary algorithm. Section 2.5 details our experimental results obtained on a popular time series forecasting benchmark. Finally, Section 2.6 gives a conclusion and introduces further research opportunities.

2.2 Related Work

2.2.1 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., 2021b). 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 chapter we apply our framework to optimize the hidden layers for a given time series forecasting task (see Figure 2.5). 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., 2020).

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. (2020) 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.

2.2.2 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. neighborhoods, 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 Figure 2.1. 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., 2018a, 2019a). 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., 2024). For hierarchical encodings, they are bundled in blocks (Pham et al., 2018; Shu et al., 2020; Liu et al., 2018b; 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., 2018). 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.

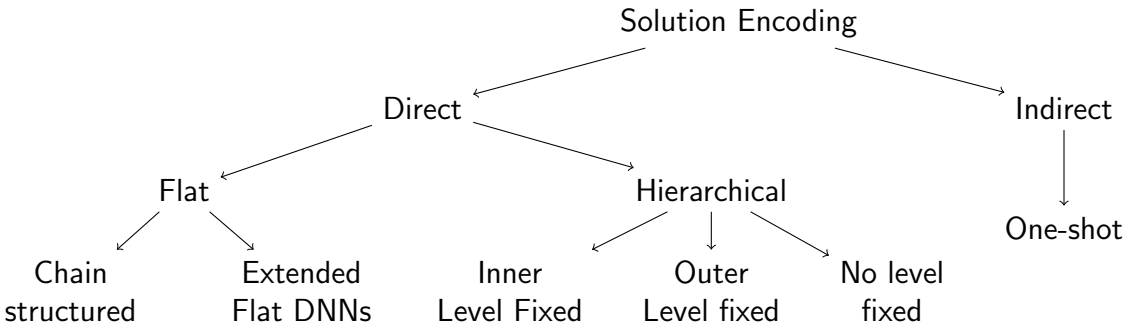


Figure 2.1: Classification of encoding strategies for NAS (Talbi, 2021).

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. (2018d). 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., 2019a), 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., 2018d), 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. (2024) 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 chapter 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.

2.2.3 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 chapter, only tackles the model selection and hyperparam-

eters 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.

2.3 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 and then we detail DRAGON's search space and its characteristics.

2.3.1 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 $\mathscr{D}$. For any subset $\mathscr{D}_0 = (X_0, Y_0) \in \mathscr{P}(\mathscr{D})$, where $\mathscr{P}(\mathscr{D})$ is the powerset of $\mathscr{D}$, we define the objective function ℓ as:

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

The explicit formula for ℓ depends on the dataset at hand and will be given later for the experiments from this chapter. Each element f from Ω is a DNN defined as an operator parameterized by three parameters. First, its architecture $\alpha \in \mathscr{A}$. The search space of all considered architectures is called $\mathscr{A}$ and will be detailed in SubSection 2.3.2. 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 2.3.3. Finally, α and λ generate an ensemble of possible weights $\Delta(\alpha, \lambda)$, from which the DNN optimal weights $\hat{\delta}$ 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 $\mathscr{D} = (X, Y)$ is composed of a target variable $Y = \{\mathbf{y}_t\}_{t=1}^{N}$, with $\mathbf{y}_t \in \mathbb{R}^H$ the target value at the time step t, and a set of explanatory variables (features) $X = \{\mathbf{x}_t\}_{t=1}^{N}$, with $\mathbf{x}_t \in \mathbb{R}^{F_1 \times F_2}$. The size of the target Y at each time step is H, 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 $H = 1$. We partition our time indexes into three groups of successive time steps and split accordingly $\mathscr{D}$ into three

datasets: $\mathcal{D}_{\text{train}}$, $\mathcal{D}_{\text{valid}}$ and $\mathcal{D}_{\text{test}}$.

After choosing an architecture α and a set of hyperparamaters λ, we build the DNN $f^{\alpha,\lambda}$ and use $\mathcal{D}_{\text{train}}$ to train $f^{\alpha,\lambda}$ and optimize its weights δ by stochastic gradient descent with respect to the training loss ℓ_{train} (the training loss and the objective function might be different depending on the use case):

$$\hat{\delta} \in \underset{\delta \in \Delta(\alpha,\lambda)}{\operatorname{argmin}} \big(\ell_{\text{train}}(f_{\delta}^{\alpha,\lambda}, \mathcal{D}_{\text{train}})\big).$$

We use a forecast error of the DNN parameterized by $\hat{\delta}$ on $\mathcal{D}_{\text{valid}}$ as our objective function to assess the performance of the selected α and λ. The best architecture and hyperparameters are optimized by solving:

$$(\hat{\alpha}, \hat{\lambda}) \in \underset{\alpha \in \mathcal{A}}{\operatorname{argmin}} \Big(\underset{\lambda \in \lambda(\alpha)}{\operatorname{argmin}} \big(\ell(f_{\hat{\delta}}^{\alpha,\lambda}, \mathcal{D}_{\text{valid}})\big)\Big).$$

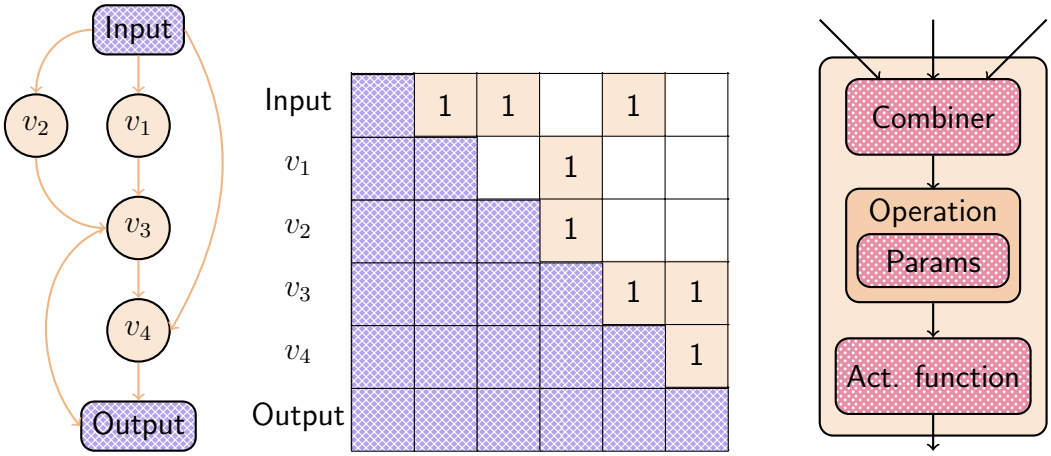
The function $(\alpha, \lambda) \mapsto \ell(f_{\hat{\delta}}^{\alpha,\lambda}, \mathcal{D}_{\text{valid}})$ corresponds to DRAGON's objective function. We finally will evaluate the performance of DRAGON by computing the objective function on $\mathcal{D}_{\text{test}}$ using the DNN with the best architecture,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

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

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 optimization sequences during which only the architecture is optimized, and others during which only the hyperparameters are optimized. In the following, we will describe our search space $\Omega = (\mathcal{A} \times \{\lambda(\alpha), \alpha \in \mathcal{A}\})$.

2.3.2 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 Section 2.2.1, 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,



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

Figure 2.2: 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).

several nodes' ordering may be valid for the same graph. For example Figure 2.2a, the orderings $v_1 > v_2$ and $v_2 > v_1$ are both valid.

Hence, a DAG Γ is represented by a sorted list $\mathbf{L}$, such that $|\mathbf{L}| = m$, containing the graph nodes, and its adjacency matrix $\mathbf{M} \in \{0, 1\}^{m \times m}$ (Zhang et al., 2019). The matrix $\mathbf{M}$ is built such that: $\mathbf{M}(i, j) = 1 \Leftrightarrow (v_i, v_j) \in \mathcal{E}$. 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 $\sum_{i=1}^{m} \mathbf{M}_{i,1} = 0$ and $\sum_{j=1}^{m} \mathbf{M}_{m,j} = 0$. Besides, the input is necessarily connected to the first node and the last node to the output for any graph, enforcing: $\mathbf{M}_{1,2} = 1$ and $\mathbf{M}_{m-1,m} = 1$. 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: $\forall i < m : \sum_{j=i+1}^{m} \mathbf{M}_{i,j} > 0$ and $\forall j > 1 : \sum_{i=j+1}^{m} \mathbf{M}_{i,j} > 0$. Finally, the ordering of the partial nodes does not allow a bijective encoding: several matrices $\mathbf{M}$ may encode the same DAG.

To summarize, we have $\mathcal{A} = \{\Gamma = (\mathcal{V}, \mathcal{E}) = (\mathbf{L}, \mathbf{M})\}$. The graphs Γ are parameterized by their size m which is not equal for all graphs. As we will see in Section 2.4.1 the DNNs size may vary during the optimization.

2.3.3 Hyperparameters Search Space

For any fixed architecture $\alpha \in \mathcal{A}$, 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 2.2c). Each layer operation is associated with a specific set of parameters, like output or hidden dimensions, convolution kernel

size or dropout rate. We provide Appendix A.1 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., 2018d). 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 $\mathbf{L}$. 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 Figure 2.2c, each node ends with an activation function. The hyperparameters optimized for each node can be found Appendix A.1.

To summarize, we define every node as the sequence of combiner $\rightarrow$ layer $\rightarrow$ 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 $\mathbf{L}$ is a variable-length list containing each node.

2.4 Search Algorithm

The search space from DRAGON $\Omega = (\mathcal{A} \times \{\Lambda(\alpha), \alpha \in \mathcal{A}\})$ 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 2.5 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.

2.4.1 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., 2023). 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. (2018c). 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 T, 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 Figure 2.3.

DRAGON's search space defined Section 2.3 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.

2.4.2 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

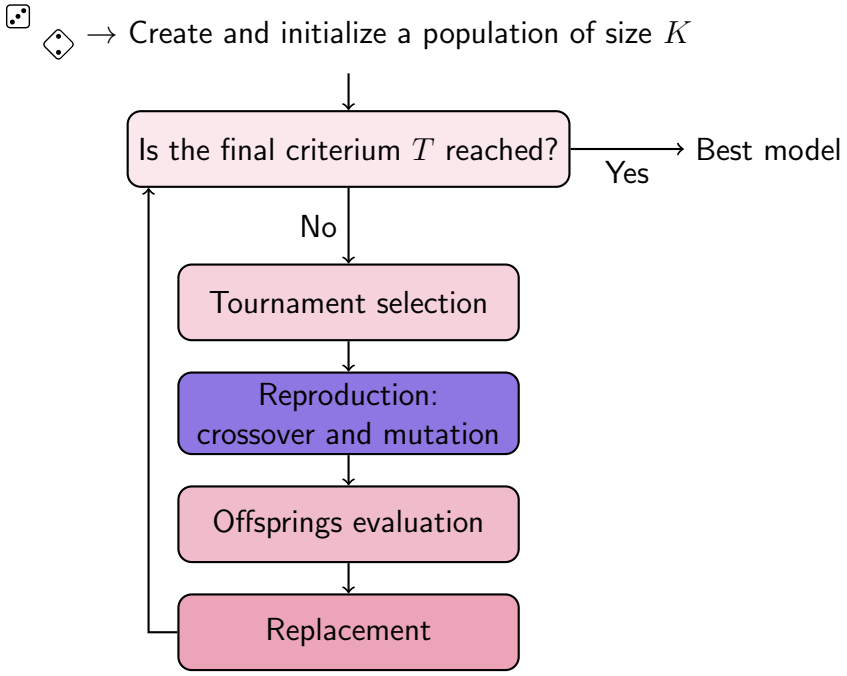


Figure 2.3: Evolutionary algorithm flowchart.

the edges between them.

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 $\Gamma = (\mathbf{L}, \mathbf{M})$, the mutation operator will draw the set $\mathbf{L}' \subseteq \mathbf{L}$ and apply a transformation to each node of $\mathbf{L}'$. Let's have $v_i \in \mathbf{L}'$ 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 2.2. 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 a 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 2.4a). 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 Figure 2.4b. 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, $v_1 \in \text{argmin}_{v \in \Gamma_1} p(v, \Gamma_1)$. The position $p(v_1, \Gamma'_2)$ of v_1 in the graph Γ'_2 can be computed as:

$$p(v_1, \Gamma'_2) \in \underset{p \in P(\Gamma'_2)}{\text{argmin}}(|p - p(v_1, \Gamma_1)|)$$

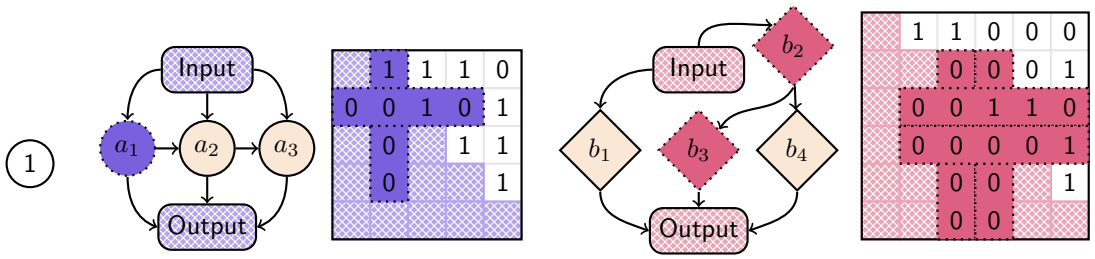
The positions from the following nodes $\{v_2, ..., v_g\} \in \Gamma_1$ are computed to respect the structure of Γ_1 and Γ_1:

$$p(v_i, \Gamma'_2) = \min\left(p(v_i, \Gamma_1) - p(v_{i-1}, \Gamma_1) + p(v_{i-1}, \Gamma'_2), |\Gamma'_2| + |\Gamma_1|\right)$$

Finally, as shown Figure 2.2a, 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.

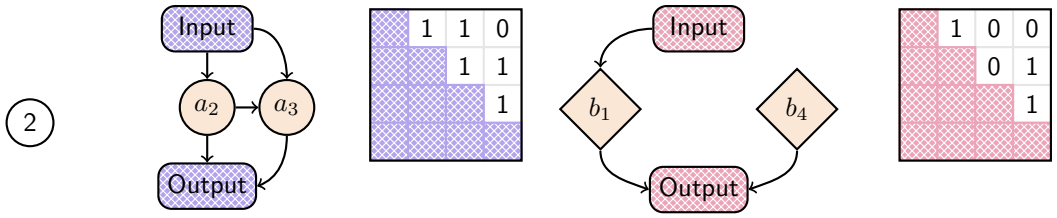
2.4.3 Hyperparameters Evolution

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 $\mathbf{L}' \subseteq \mathbf{L}$ and apply a transformation on each node of $\mathbf{L}'$. For each node v_i from $\mathbf{L}'$, we draw h_i hyperparameters, which will be modified by a neighboring value. The hyperparameters in our search space



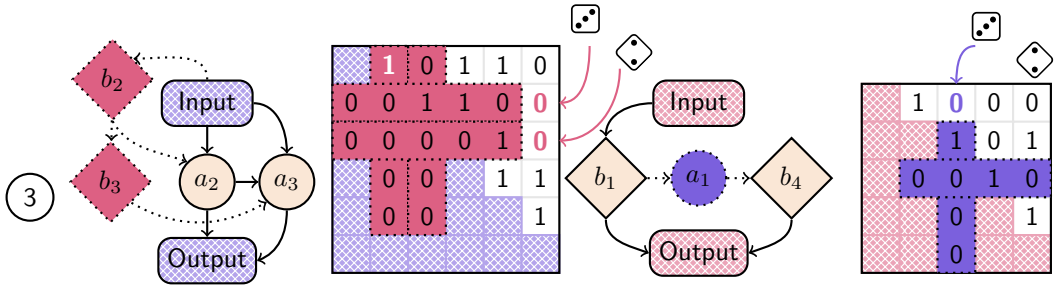
Parent 1: Γ_1 Parent 2: Γ_2

(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.



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.



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 2.4: Crossover operator illustration.

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 neighbors 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 neighbors inside a continuous interval around the actual value. Such a neighborhood has been defined for instance to the dropout rate.

2.5 Experimental Study

In this section, we describe how we evaluated DRAGON on a time series forecasting task. In the first three sections (Section 2.5.1, Section 2.5.2 and Section 2.5.3), 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 (Section 2.5.4, Section 2.5.5 and Section 2.5.6) we present and analyze the results. Finally, the last two sections (Section 2.5.7 and Section 2.5.8) discuss the limitations of the work and give some hints for further work.

2.5.1 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 optimized 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[1] 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

[1] https://forecastingdata.org/

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.

2.5.2 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 Appendix A.2. 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 $T = 8$ 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 (ℓ_{train}, a mean squared error in this case) 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. The objective function ℓ in this case is then 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:

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

In our case, for $f \in \Omega$, $\mathcal{D}_0 = (X_0, Y_0) \subseteq \mathcal{D}$, we have:

$$\ell\big(Y_0, f(X_0)\big) = \mathrm{MASE}\big(Y_0, f(X_0)\big).$$

2.5.3 Search Space

The generic search space defined Section Section 2.3 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 2.5, 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 Appendix A.1. 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

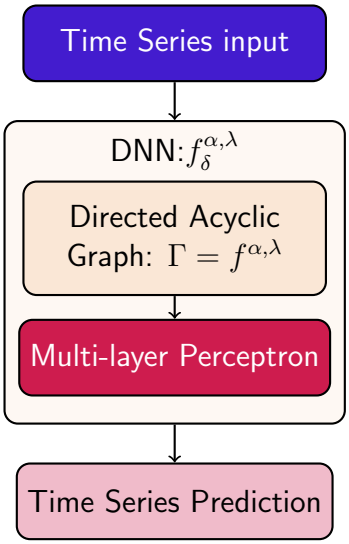


Figure 2.5: Meta-architecture for Monash time series datasets.

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.

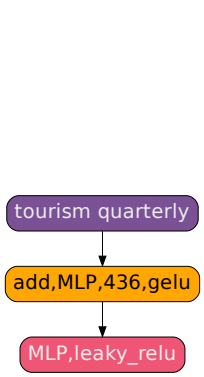
2.5.4 Results

We report a summary of the results Table 2.1. 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 AutoDL framework, was only able to beat it on 7 datasets out of 27 (26%). More

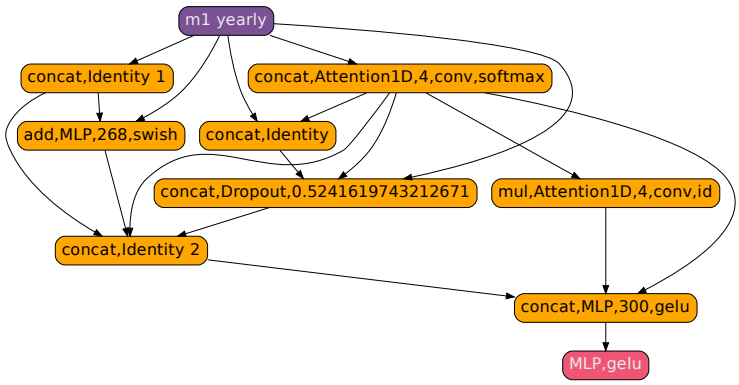
Algorithm(s)	Wins	Losses	Champion	Failures
SES	1	25	0	1
Theta	6	20	0	1
TBATS	6	20	0	1
ETS	8	18	1	1
(DHR-)ARIMA	4	21	0	2
PR	1	25	0	1
CatBoost	1	25	0	1
FFNN	0	26	0	1
N-BEATS	0	26	0	1
WaveNet	2	23	0	2
Transformer	0	26	0	1
DeepAR	1	25	1	1
TFT	4	23	0	0
SeasonalNaive	1	26	0	0
Prophet	2	24	1	1
AutoPytorch	7	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 2.1: 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.

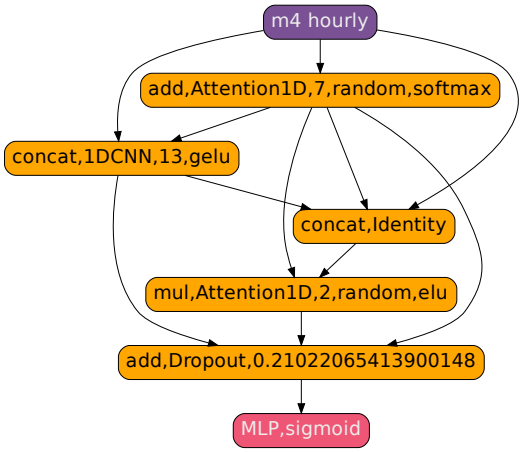
detailed results can be found Table 2.2, and visual representations of the DNNs found for some time series can be found Figure 2.6. The content of the Γ graph is shown in yellow, while the last MLP layer is shown in pink.



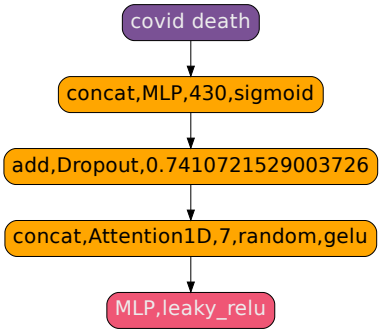
(a) Tourism Quarterly



(b) M1 Yearly



(c) M4 Hourly



(d) Covid Death

Figure 2.6: 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, $\mathcal{S}$ the set of the 22 algorithms from the baseline and $l_{p,s}$ the final score (loss) of the algorithm $s \in \mathcal{S}$ on the dataset $p \in \mathscr{P}$. We define the performance ratio $r_{p,s}$ of s on p as:

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

From this we can define the performance profile as the probability for the algorithm

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 2.2: 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.

$s \in \mathcal{S}$ 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 2.7. From the performance profile, we can see that compared to

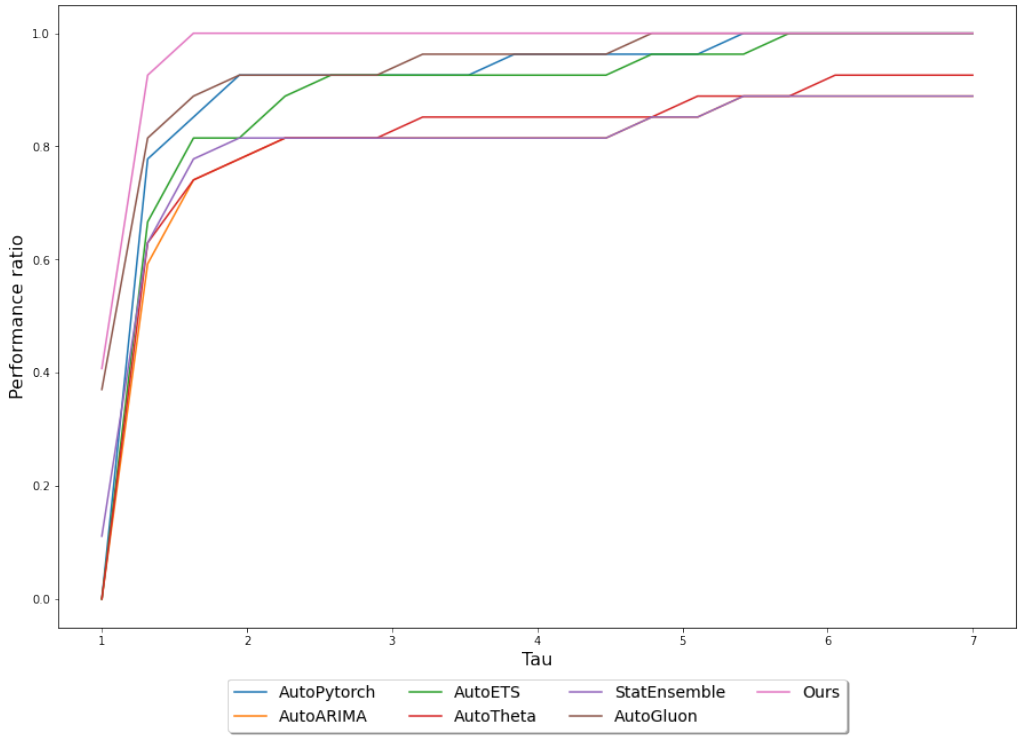


Figure 2.7: Performance profile $\rho_s(\tau)$ for each algorithm s from the AutoML baseline, with $\tau \in [1, 7]$.

the baseline, 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.

2.5.5 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 2.8 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

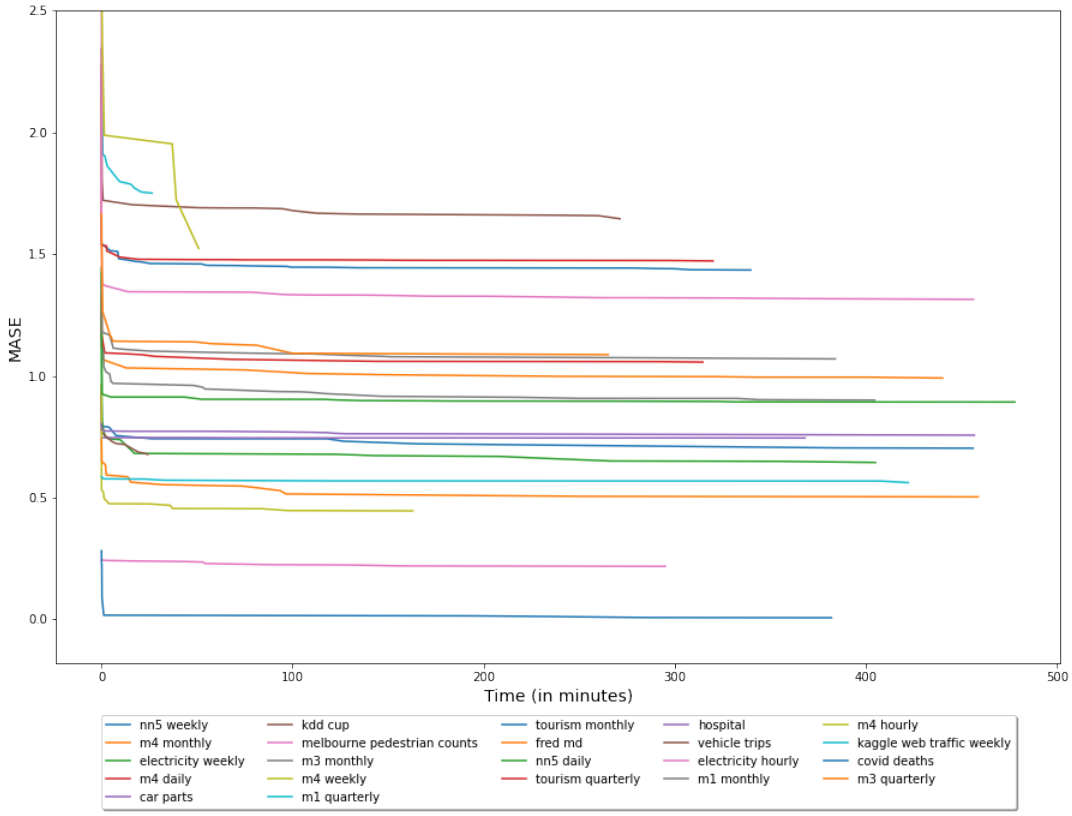


Figure 2.8: Computation time of DRAGON for each dataset. The curves represent the time when the best loss so far has been found for each dataset.

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 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.

2.5.6 Best Models Analysis

In the AutoDL literature, little effort is usually made to analyze the generated DNNs. Shu et al. (2020) 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 2.3:

- **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 2.3 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 2.6a. 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 2.3) is the most frequent layer, as it appears on average at least once per graph, although more complex layers such as convolution, recurrence

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

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?

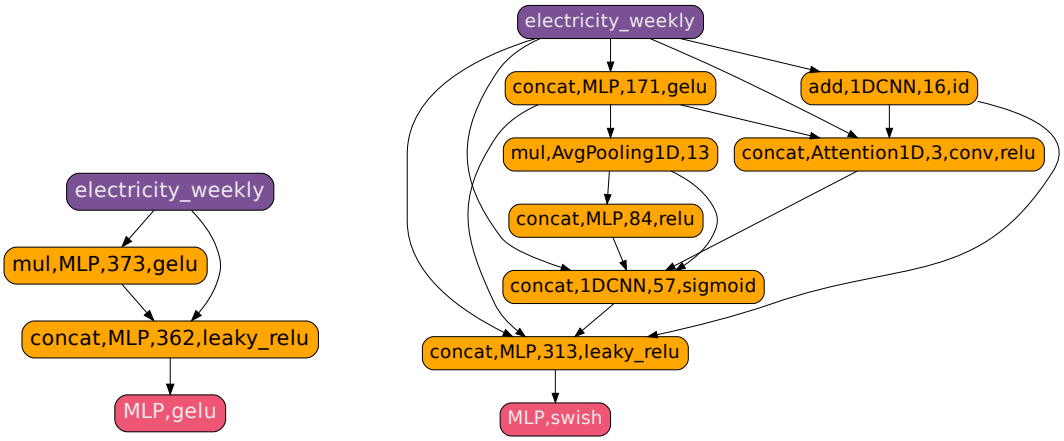
The structural indicators for all datasets from Table 2.3 are significantly different for each dataset, which means that the framework does not converge to similar architectures. As we set the seed, the initial population of size K is identical for each dataset, but then the performance of the evaluated models affects the creation of the following graphs, leading to different final models optimized for each time series. Furthermore, Figure 2.9 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 2.3 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 et al. (2020)?

To answer this question, the observations from Shu et al. (2020) 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 2.5. 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 2.9: Two different models having similar good performance on the Electricity Weekly dataset (best MASE: 0.644).

2.5.7 Ablation Study

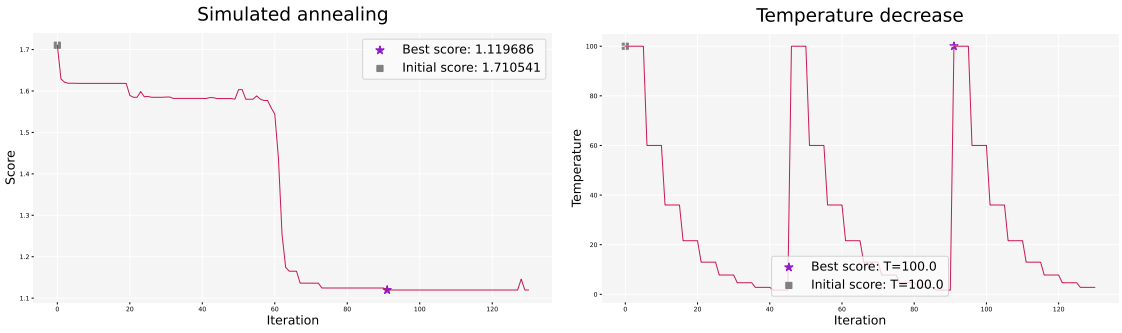
We chose two datasets, M1 monthly and Tourism monthly, in order to reduce the number of experiments we had to perform, as the benchmark was quite large. We compared four search algorithms for both datasets; these were random search, a population based evolutionary algorithm (EA) with alternating optimization of hyperparameters and architecture, as well as a version with joint optimization, and, lastly, simulated annealing. To explore the search space, we used an exponential multiplicative monotonic cooling schedule in our simulated annealing algorithm: $\mathrm{Temp}_k = \mathrm{Temp}_0.\alpha^k$. 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 Table 2.4.

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

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

simulated annealing algorithms from their starting points. The Figure 2.10 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. We compared the convergence of both algorithms using Figure 2.11. We note that the mean loss of the populations is more stable in the case of joint mutation, without the two phases present in the alternate version. Additionally, the alternate version converges a bit faster towards a the final solution.



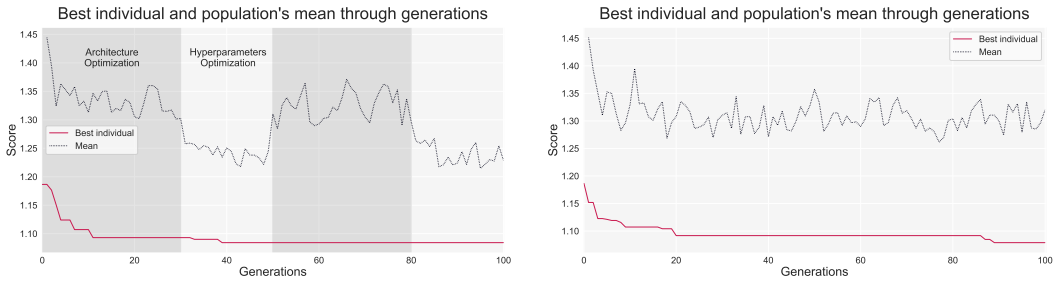
(a) Best score for each iteration of the simulated annealing algorithm.

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

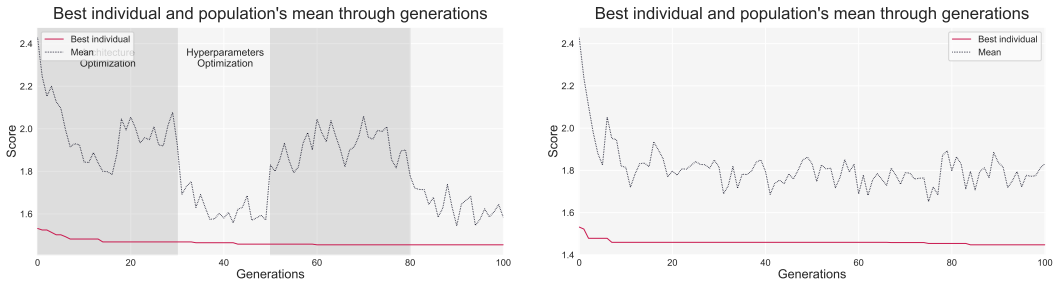
Figure 2.10: Simulated annealing algorithm for the M1 Monthly dataset with seed=100, MASE=1.120.

2.5.8 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



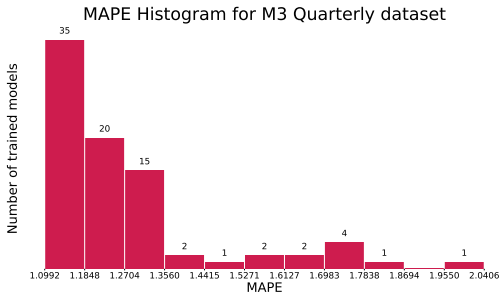
(a) M1 Monthly dataset, with seed=400. Left: MASE=1.084, right: MASE=1.073.



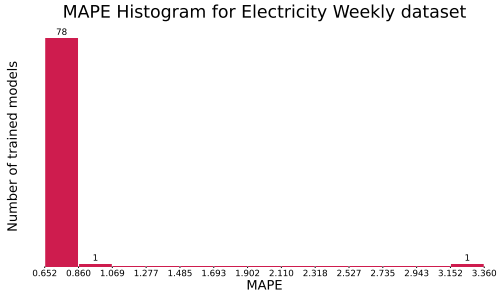
(b) Tourism Monthly dataset, with seed=200. Left: MASE=1.453, right: MASE=1.448.

Figure 2.11: Evolutionary algorithm's population mean loss and best individual's loss through generations. Left: the mutation operator alternate between optimizing only the architecture, then the hyperparameters. Right: the mutation operator jointly optimize the architecture and hyperparameters.

and set of hyperparameters can produce a large variety of results on a dataset. Figure 2.12 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 2.12. 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 2.12 showed that the optimization was effective as no other seeds gave better results than the one picked by DRAGON. However, since the seed does not contain any optimizable information, trying to tune it can lead to overfitting. In order to avoid this type of problem, the rest of the manuscript presents other solutions for stabilizing the neural networks found and reduce the seed's effects on the performance.



(a) M3 Quarterly, best MASE: 1.099



(b) Electricity Weekly, best MASE: 0.652

Figure 2.12: MASE histogram of the best model performances with multiple seeds for two datasets.

2.6 Conclusion and Future Work

In this chapter, we introduce 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 our framework 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, our framework shows superior forecasting capabilities 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 our framework 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.

Chapter 3

A Bandit Approach With Evolutionary Operators for Model Selection: Application to Neural Architecture Optimization for Image Classification

In this chapter, we are introducing a new algorithm that is compatible with our search space and more efficient than the evolutionary algorithm used in the previous search space. Our search space includes graph structures that are difficult to handle for most efficient search algorithms, such as Bayesian optimization or gradient-based methods. Given the design of the neighborhoods, the evolutionary algorithm was the obvious choice to navigate this space. However, this algorithm is slow because it gives the same amount of resources to every configuration, even if they are not promising. The proposed algorithm of this chapter, named Mutant-UCB, integrates a bandit-based approach with evolutionary operators, leveraging the flexibility of the evolutionary algorithm while enhancing resource allocation and selection processes. To illustrate the versatility of Mutant-UCB, the algorithm is presented in the generic context of model selection, with experiments conducted within our search space to optimize deep neural networks (DNNs) for image classification. The efficacy of Mutant-UCB is demonstrated by its superior performance in comparison to a random search, an evolutionary algorithm, and the bandit-based algorithm Hyperband, as evidenced by experiments conducted on three open-source datasets. It is important to note that in this chapter, the variable t does not refer to a timestamp within a time series, but rather to an algorithm iteration. The variable N will refer to a number of training instances, not a number of samples. Finally, the other chapters focus on minimizing an objective loss function, while this chapter addresses maximizing an accuracy function.

Keisler, J. and Brégère M. **A Bandit Approach With Evolutionary Operators for Model Selection: Application to Neural Architecture Optimization for Image Classification.** *International Conference on Knowledge Discovery and Data Mining (ACM SIGKDD), International Workshop on Resource-Efficient Learning for Knowledge Discovery*, 2024.

Contents

3.1 Introduction

Accuracy of machine learning models significantly depends on some parameters which cannot be modified during training. As the number of parameter combinations to be tested exponentially increases with the number of these parameters, it becomes costly and time-consuming to optimize them. Automating the selection of promising models, usually referred to as AutoML (Automated Machine Learning), is a fast-growing area of research (see Hutter et al. 2019 for a quite recent book). We approach the model selection problem in a general manner without any restrictions on the nature of the hyper-parameters, such as the types of machine learning models, neural network architectures, or hyper-parameters of random forests. Our aim is to find the best model without making any assumptions about the task, model type, or reward to maximize. We assume that we have access to an infinite number of possible models and set a predetermined budget of resources T, used to train the models. These resources are allocated to the models in the form of "sub-trains", such as iterations, data samples or features. The final (and hopefully the best) model is chosen by finding a good trade-off between

exploration (training a large number of models) and exploitation (allocating a large budget to promising models). This process may fall under the umbrella of multi-armed bandits (see Lattimore and Szepesvári 2020 for an in-depth review).

In this chapter, we treat model selection as an instance of best-arm identification in infinite-armed bandits. We propose a new model selection algorithm, called Mutant-UCB: an Upper Confidence Bound (UCB)-based algorithm (see Auer et al. 2002 for UCB's original idea) that incorporates a mutation operator from the evolutionary algorithms. This operator creates a new model from the neighborhood of the model selected by the bandit algorithm. Unlike most model selection algorithms, Mutant-UCB makes no assumptions about the solutions encoding, also called search space, or the reward function to be maximized, making it suitable for a wide range of configurations. The use of a UCB-type algorithm and adaptive resource allocation allows exploration of the search space, while the mutation operator effectively directs the search towards promising solutions. Results on a neural networks optimization problem demonstrate the relevance of this approach.

We begin this chapter by presenting the setup of our bandit model selection approach in Section 3.2 and we position ourselves in relation to the state of the art. Mutant-UCB, the algorithm we develop, is presented in Section 3.3. Section 3.4 is dedicated to experiments on the optimization of deep neural networks: we validate the performance of Mutant-UCB on three open-source image classification datasets. Finally, Section 3.5 discusses the advantages of Mutant-UCB compared to the state of the art and opens new research perspectives.

3.2 Model selection problem setup: a bandit approach

3.2.1 Literature Discussion

Naive strategies for model selection are Grid or Random Search. More sophisticated strategies address model selection as a sequential learning problem. Two approaches stand out: **configuration selection** methods sequentially select new models ("close" to promising models) to train, while **configuration evaluation** methods allocate more resources (training time) to promising models. The first approach suggests that accuracy is regular with respect to some distance between models implying the existence of an underlying space. Therefore, two models that are close to each other will have similar performance. The second approach, on the other hand, makes no assumptions about the potential (smooth) links between model performances.

Evolutionary methods. Among the configuration selection methods, evolutionary algorithms have been popular for many years (see, e.g., Young et al. 2015 and Jian et al. 2023). Starting from an initial set of configurations, they evolve them towards performing models using unitary operators like the mutation (little change in the configuration).

Even complex operators involving more than two configurations like the crossover are considered by Strumberger et al. (2019). These algorithms are highly versatile and can be applied to a wide range of setups. The literature presents different methods that vary in terms of search spaces, i.e. the way configurations are encoded. The operators used to generate the new population typically depend on this encoding. Usually, the configurations are represented as character strings or lists and can be modified using bit-string mutations and combined with k-point crossovers (see Eiben and Smith 2015 for more details). But recent works, mostly for neural networks architecture optimization, tried to design other representations and operators. For instance a tree-based mutation operator to optimize recurrent neural networks is proposed by Rawal and Miikkulainen (2018). Awad et al. (2021) use differential evolutionary operators to optimize neural network hyper-parameters and architectures. One disadvantage of the evolutionary algorithms is the large number of parameters involved, such as the population size, the selection function, or the elitism rate. Choosing the appropriate values for these parameters can be complex.

Bayesian optimization. Bayesian optimization has recently emerged as a more efficient approach than evolutionary methods in AutoML (see, among others, Malkomes et al. 2016, Zoph and Le 2016 and Kandasamy et al. 2018). It is a sequential optimization technique commonly used to minimize black-box functions. Those algorithms are based on two main components, a surrogate model that approximates the unknown black-box function, and an acquisition function that selects the next element in the search space to be evaluated. One major limitation of these acquisition functions is their reliance on strong assumptions about the black-box function and the search space (see Garrido-Merchán and Hernández-Lobato 2020 for further details). Therefore, we did not employ a Bayesian optimization algorithm in our experiments as we aimed to avoid making any assumptions about the smoothness, distance or continuity of the search space or the reward function.

Bandits approaches. Firstly, still in the field of Bayesian optimization, the extensions GP-UCB and KernelUCB (see, Srinivas et al. 2010 and Valko et al. 2013, respectively) of the classical UCB bandit algorithm and more recent algorithms largely inspired by them (see, e.g., Dai et al. 2024) have been massively used for optimization and eventually model selection. The BayesGap algorithm introduced by Hoffman et al. (2014) connects Bayesian optimization approaches and best arm identification, assuming correlations among the arms. More recently, Huang et al. (2021) sees the neural architecture search as a combinatorial multi-armed bandit problem which allows the decomposition of a large search space into smaller blocks where tree-search methods can be applied more effectively and efficiently. Configuration evaluation approaches have also been investigated in an infinite or multi-armed bandit framework. At each iteration of the algorithm, a new arm/model can be drawn from an infinite search space containing the models and added

to the set of models already (more or less) trained. Karnin et al. (2013) proposes the Sequential (or Successive) Halving algorithm, which splits the given budget evenly across an optimal number of elimination rounds, and within a round, pulls arms in a uniform manner. It comes with solid theoretical guarantees that have recently been improved by Zhao et al. (2023). Li et al. (2018) proposes the Hyperband algorithm, a robust extension of Sequential Halving, and applies it to deep neural networks hyperparameters optimization. Moreover, Shang et al. (2019) introduces D-TTTS, an algorithm inspired by Thompson sampling. Hybrid methods combine adaptive configuration selection and evaluation: Terayama et al. (2021) proposes a rule to stop training a model prematurely based on the predicted performance from Gaussian Process; in addition, Kandasamy et al. (2016) extends GP-UCB to enable sequential model training (and thus resource allocation).

Best-arm identification in infinite-armed bandits. The stochastic infinite-armed bandit framework has been introduced and studied for the cumulative reward maximization problem by Berry et al. (1997) and Wang et al. (2008). Carpentier and Valko (2015) and Aziz et al. (2018) study best armed identification problem in this framework. Theoretical results attest to the performance of their strategies (SiRI and extensions; α,ϵ-KL-LUCB, respectively).

3.2.2 Contributions

The main contribution of this study is the Mutant-UCB algorithm, which incorporates operators from evolutionary algorithms into the UCB-E (Upper Confidence Bound Exploration) algorithm introduced by Audibert et al. (2010). It combines both configuration evaluation and configuration selection approaches: it is sequential in computation and picks a (generally promising) model thanks to a UCB-based criteria. Then it either continues its training (resource allocation) or creates and starts training a new model derived from the selected one thanks to the "mutation" operation of an evolutionary algorithm. This last possibility is based on the intuition that the expected "mutant model" accuracy will be close to that of the original model. While bandit approaches have been used to design the "selection" operator for evolutionary algorithms by Li et al. (2013), to our knowledge this is the first time that operators from evolutionary algorithms are incorporated into a bandit algorithm.

Afterwards, we compare Mutant-UCB to a Random Search, the evolutionary algorithm proposed Chapter 2 and the Hyperband algorithm introduced by Li et al. (2018) on three open source data sets collected for image classification: CIFAR-10 (Krizhevsky et al., 2009), MRBI (Larochelle et al., 2007) and SVHN (Netzer et al., 2011). For a fair comparison, Mutant-UCB and the evolutionary algorithm under consideration share the same mutation operation.

3.2.3 Set-up

In the infinite-armed bandit framework, when a new arm k is pulled from the reservoir, the expectation of the accuracy of the associated model μ_k pulled from the search space is assumed to be an independent sample from a fixed distribution. With T a fixed budget, at each round $t = 1, \dots, T$, an arm I_t is picked and a sub-train (allocation of a resource) is performed on the associated model. This model is then evaluated on a validation data set $\mathcal{D}_{\text{valid}}$ using an accuracy function acc we aim to maximize. This accuracy corresponds to the reward a_t. After T rounds, we select the final arm $\widehat{I}_T$. In what follows, the untrained model associated with arm k is denoted f_k, and after N_k sub-trains we denote it $f_k^{N_k}$.

3.3 UCB-based algorithm incorporating mutation operators from evolutionary algorithms

3.3.1 A brief reminder of the UCB-E algorithm

Designed for best arm-identification in a K-multi-armed bandit problem, the UCB-E algorithm proposed by Audibert et al. (2010) is recalled in Algorithm 1. This highly exploratory policy is based on the principle of optimism in the face of uncertainty, in the spirit of the UCB algorithm introduced by Auer et al. (2002). It aims to find the best model among K untrained models $f_1, \dots, f_K$ sampled from the search space. The algorithm starts by K rounds of deterministic exploration: it performs a first sub-train per model and observes the accuracies $a_k = \text{acc}\big(f_k^1, \mathcal{D}_{\text{valid}}\big)$. At each round $t = K+1, \dots, T$, and for each k, it computes the empirical mean accuracy $\hat{\mu}_{k,t}$ from the previous rewards associated with arm k:

$$\hat{\mu}_{k,t} = \frac{1}{N_{k,t}} \sum_{s=1}^{t-1} a_t \mathbf{1}_{I_s=k} \quad \text{with} \quad N_{k,t} = \sum_{s=1}^{t-1} \mathbf{1}_{I_s=k}\,. \tag{3.1}$$

Then, it chooses the arm optimistically:

$$I_t \in \underset{k\in\{1,\dots K\}}{\text{argmax}} \left\{ \hat{\mu}_{k,t} + \sqrt{\frac{E}{N_{k,t}}} \right\}, \tag{3.2}$$

performs a sub-train on the associated model and receives the reward:

$$a_t = \text{acc}\big(f_{I_t}^{N_{I_t,t+1}}, \mathcal{D}_{\text{valid}}\big).$$

For the sake of readability, the iteration index for the counting N_k and empirical mean μ_k variables in Algorithms 1 and 2 have been removed. These variables are updated

throughout the iterations.

The core issue is the tuning of the exploration parameter E. Audibert et al. (2010) show that the optimal value depends on the difficulty of the underlying bandit problem, which has no reason to be known in advance.

Algorithm 1: $UCB - E$

Inputs:
T budget
E exploration parameter
K number of untrained models

Initialization
Sample K untrained models $f_1, \dots, f_K$
For $k = 1, 2, \dots, K$
Perform a first sub-train on f_k which becomes f_k^1
Get the reward $a_k = \mathrm{acc}\left(f_k^1, \mathcal{D}_{\text{valid}}\right)$
Define $N_k = 1$, $\hat{\mu}_k = a_k$

For $t = K+1, K+2, \dots, T$
Choose $I_t \in \mathrm{argmax}_{k \in \{1,\dots K\}} \left\{ \hat{\mu}_k + \sqrt{\frac{E}{N_k}} \right\}$
Perform a sub-train: model $f_{I_t}^{N_{I_t}}$ becomes $f_{I_t}^{N_{I_t}+1}$
Get the reward $a_t = \mathrm{acc}\left(f_{I_t}^{N_{I_t}+1}, \mathcal{D}_{\text{valid}}\right)$
Update $\hat{\mu}_{I_t} = \frac{1}{N_{I_t}+1}\left(a_t + N_{I_t}\hat{\mu}_{I_t}\right)$ and $N_{I_t} = N_{I_t} + 1$

Output:
Model $f_{\widehat{I}_T}^{N_{\widehat{I}_T}}$ where $\widehat{I}_T \in \mathrm{argmax}_{k \in \{1,\dots K\}} \hat{\mu}_k$

3.3.2 Main contribution: the Mutant-UCB algorithm

Mutant-UCB, presented in Algorithm 2, incorporates two main ideas into UCB-E. First of all, there is no point in multiplying the number of sub-trains for the same model: there generally comes a time when it is no longer useful, so we can potentially define a maximum number of sub-trains N. Note that this idea of a maximum quantity of resources that can be allocated to a single model was already present in the Hyperband algorithm proposed by Li et al. (2018). Furthermore, in model selection, it is not uncommon for similar models to perform similarly, which is why configuration selection methods may be so effective in this task. In general, the main problem lies in defining a distance between models: search spaces are usually high-dimensional and hyper-parameters are of various kinds (learning rate, type of activation function, number of neurons, etc.). While the notion of distance between two models is not easy to define, evolutionary algorithms offer a good compromise: they breed new individuals through crossover and mutation operations. Crossover operations mix two individuals, while mutation operations can be applied to a single individual in order to create "mutants", involving some tiny changes.

Those "mutants" can be seen as neighbors of the initial point. We could therefore imagine that a model chosen by the algorithm could mutate to give rise to a new one, with the intuition that the mutant and its original model will have similar accuracies. To our knowledge, the inclusion of mutation operators of evolutionary algorithms in a bandit algorithm is completely new.

Like the UCB-E algorithm, Mutant-UCB starts with the first sub-train of K models. At each round $t = K + 1, \ldots,$ it still chooses the next arm optimistically, by resolving Equation (3.2). For an arm k, we recall that $N_{k,t}$ is the number of times the arm has been picked before round t - see Equation (3.1). We now introduce $\overline{N}_{k,t}$, the integer that counts the number of times the model associated with arm k has been trained. Once arm I_t is picked, with $p_t = 1 - \overline{N}_{I_t,t}/N$:

- a sub-train is performed on $f_{I_t}^{\overline{N}_{I_t,t}}$ with probability p_t or
- a mutation is performed on $f_{I_t,t}^{\overline{N}_{I_t,t}}$ with probability $1 - p_t$.

The mutation is performed on the trained model $f_{I_t}^{\overline{N}_{I_t,t}}$ - and not just f_{I_t} - to include the case where certain parameters of the model optimized during training (e.g., weights of neuron networks) are passed on to its mutant. We detail the mutation operation for our use-case in Section 3.4. When a mutation occurs, a new model is created, a first sub-train is performed and the model is added to the list of potential models to be retained at the end of the algorithm. Thus, the number of models K increases by one each time a mutant model is created.

Remark 3.3.1. *When a new model comes into play, it is very likely to be quickly chosen by the algorithm, even if its accuracy is not good: the algorithm must explore this new possibility. The "sleeping bandit" framework, in which new arms may be added and/or become unavailable during the algorithm execution, is studied by Kleinberg et al. (2010). It proposes a very natural extension of UCB: the Awake Upper Estimated Reward algorithm and shows there is no need to adapt the confidence bounds.*

The probability p_t decreases as the model goes along its sub-trains and guarantees that it will not be trained more than N times. The more the model has been trained, the more likely it is to mutate when selected. The underlying idea is that further training will probably have little effect or even over-fit in the case of neural networks, and that if the algorithm selects this already well-trained model, it is because it may have good accuracy (and it will probably be the same for a mutant model). Note that the probability p_t is linear in $N_{I_t,t}$; this choice is arbitrary and we could quite easily have chosen another type of relationship, e.g., $p_t = 1 - \exp(N_{I_t,t} - N)$. The algorithm ends with a finalization phase: the best model, in terms of average accuracy, $\widehat{I}_T$ is selected among the initial models and the mutant models and its training is completed with $N - \overline{N}_{\widehat{I}_T,T-N+2}$ additional sub-trains.

Remark 3.3.2. *The idea of integrating mutation operations comes from an implicit assumption about the distribution of the expectation of the accuracies of mutant models,*

Algorithm 2: Mutant-UCB

Inputs:
T budget
E exploration parameter
K initial number of models
N maximum number of sub-trains that can be allocated to a single model

Initialization
Sample K untrained models $f_1, \ldots, f_K$
For $k = 1, 2, \ldots, K$
Perform a first sub-train on f_k which becomes f_k^1
Get the reward $a_k = \mathrm{acc}\big(f_k^1, \mathcal{D}_{\text{valid}}\big)$
Define $N_k = \overline{N}_k = 1$, $\hat{\mu}_k = a_k$

For $t = K+1, K+2, \ldots, (T-N+1)$
Choose $I_t \in \mathrm{argmax}_{k\in\{1,\ldots K\}} \left\{\hat{\mu}_k + \sqrt{\frac{E}{N_k}}\right\}$
Sample $X_t \sim \mathcal{B}(p_t)$ with $p_t = 1 - \overline{N}_{I_t}/N$
If $X_t = 1$:
Perform a sub-train: model $f_{I_t}^{\overline{N}_{I_t}}$ becomes $f_{I_t}^{\overline{N}_{I_t}+1}$
Get the reward $a_t = \mathrm{acc}\big(f_{I_t}^{\overline{N}_{I_t}+1}, \mathcal{D}_{\text{valid}}\big)$
Update $\hat{\mu}_{I_t} = \frac{1}{\overline{N}_{I_t}+1}\big(a_t + \overline{N}_{I_t}\hat{\mu}_{I_t}\big)$, $N_{I_t} = N_{I_t} + 1$
and $\overline{N}_{I_t} = \overline{N}_{I_t} + 1$
Else :
Update the number of models $K = K + 1$
Create a mutant model f_K from $f_{I_t}^{\overline{N}_{I_t}}$
Perform a first sub-train on f_K which becomes f_K^1
Get the reward $a_t = \mathrm{acc}\big(f_K^1, \mathcal{D}_{\text{valid}}\big)$
Define $N_K = \overline{N}_K = 1$, $\hat{\mu}_K = a_t$
Update $N_{I_t} = N_{I_t} + 1$

Finalization
Select the best model $\widehat{I}_T \in \mathrm{argmax}_{k\in\{1,\ldots K\}} \hat{\mu}_k$
Finalize its training by performing $N - \overline{N}_{\widehat{I}_T}$ sub-trains

Output: $f_{\widehat{I}_T}^N$

e.g.,

$$\mathbb{E}\Big[\, \mu_k \,\Big|\, f_k \text{ is a mutant of } f_j^{\overline{N}_j} \,\Big] = \mu_j \,.$$

Theses kind of assumption in our case study are discussed in Appendix B.3.

3.3.3 Hardware implementation

We implemented Mutant-UCB in an HPC environment, on multiple GPUs, using parallelization to allocate resources as efficiently as possible. So, if we have D GPUs, we create $D+1$ processes, one worker per GPU plus one master process. Once a d worker process is available, the master process assigns it a model f from the population using the equation 3.2. If f is to be trained, it is removed from the population and trained by the d process on its associated GPU. The f model cannot then be selected by the next d' process that presents itself to the master process. At the end of its training, the f model is reintegrated into the population, and its errors and counters are updated. Therefore, most of the time when a worker process is free, the $D-1$ others are all busy training their models. Thus, the selection of the next arm to be drawn is based on a population of size $K - D + 1$. If the selected model is mutated, the mutant is taken by the process to be trained, and the initial model is returned to the population. On average, D models are trained and evaluated in parallel.

3.4 Experiments

In this section, we evaluate the performance of the Mutant-UCB algorithm, on neural networks optimization. In order to highlight the advantages of our method, we put ourselves in a case where we make no assumptions about the smoothness of the reward function acc and we do not consider any distance between the elements f_k from our search space. We therefore compare our methods with three algorithms that are applicable in this case: a Random Search, the Hyperband algorithm and an evolutionary algorithm. This neural networks optimization is applied to three image classification data sets.

3.4.1 Experiment design

Data sets. We performed our experiments using three image classification data sets, also used by Li et al. (2018) to introduce the Hyperband algorithm: CIFAR-10 Krizhevsky et al. (2009), Street View House Numbers (SVHN Netzer et al. 2011) and rotated MNIST with background images, also called MRBI Larochelle et al. (2007). These first two data sets contain 32×32 RGB images, while MRBI contains 28×28 gray-scale images. The labels for each data set are converted to integers between 0 and 9. We split each data set into a training, a validation and a testing set. The training set is used to optimize

the model weights (namely to perform the sub-trains), while the validation set is used to evaluate the configurations in the selection model algorithms (i.e. to get the rewards). Finally, the accuracies of the configuration selected by the algorithms are computed on the testing set to assess their quality. CIFAR-10 has 35k image on the train set, 15k on the validation set and 10k on the test set, SVHN has 51k, 22k and 26k and MRBI 10k, 2k and 50k data points on the three sets respectively. For all data sets we standardized the images so the input has a mean of zero and a standard deviation of one.

Search space: the pool of possible configurations. We used the DRAGON framework introduced Chapter 2 to encode our neural networks. In this framework, neural networks are represented as directed acyclic graphs (DAGs), where the nodes represent the layers (e.g., recurrent, feed-forward, convolutional) and the edges represent the connections between them. The task on which we want to try our algorithms is image classification. To do so, we define a generic search space (the pool of possible configurations f_k) with DRAGON, dedicated to the task at hand. Any sampled configuration f_k will be made of two directed acyclic graphs. The first one processes 2D data, and can be made of 2D convolutions, 2D pooling, normalization and dropout layers. The second one consists in a flatten layer followed by MLPs (Multi-Layers Perceptrons) and normalization layers. A final MLP layer is added at the output of the model to convert the latent vector into the desired output format. The framework includes operators, namely mutations and crossovers to modify and thus optimize the graphs. The mutation operators modify the neural network architecture by adding, removing or modifying the nodes and the connections in the graph. They can also be applied within the nodes, on the neural network hyper-parameters (e.g., convolution layer kernel size or an activation function). Crossover involves exchanging parts of two graphs.

Sub-trains. We trained our neural networks using a cyclical learning rate, as proposed by Huang et al. (2022). When the learning rate is low, the neural network reaches a local minimum. Right after, the learning rate goes up again taking the model out of the local minimum. We consider in our experiments that a sub-train is one of this loop, with learning rate getting from its maximum to its minimum. We let N be the maximum number of sub-trains for a given element f_k from our search space.

Baselines. Random Search, the Evolutionary Algorithm (EA), Hyperband and Mutant-UCB have all been implemented so that they can be used with the same search space implementation. They all use the same training and validation functions to assess the neural networks performance, and share a common budget, namely T. For Random Search, we randomly select $K_{\mathrm{RS}} = T/N$ neural networks. For each of them we perform N sub-trains, resulting in T sub-trains in total. For the Evolutionary Algorithm, we implemented an asynchronous (or steady-state) version. Compared to the standard algorithm, the steady-state evolutionary algorithm of Liu et al. (2018c) enhances efficiency

on High-Performance Computing (HPC) by producing two offsprings from the population as soon as a free process is available, rather than waiting for the entire population to be evaluated. We set an initial population of size K_{EA}, where the deep neural networks are randomly initialized. We perform N sub-trains on each of these models. Then, we evolve the population using the mutation and crossover operators from Chapter 2. If a generated offspring is better than the worst model from the population, it replaces it. During the optimization procedure, we generate $T/N - K_{\text{EA}}$ offsprings and we perform N sub-trains on each, resulting in a total of T sub-trains. For Hyperband we ran the algorithm with its parameters R and η such that the total number of sub-trains is T and that each model can be trained only N times (see, Li et al. 2018 for further details).

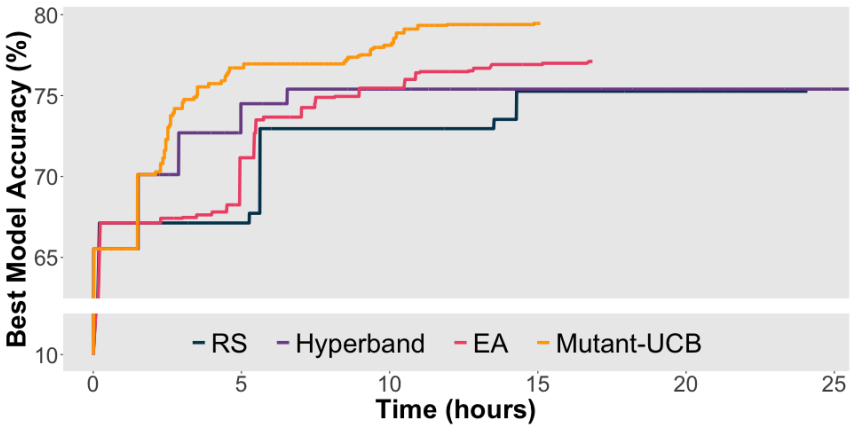
The algorithm Mutant-UCB starts with an initial population of K_{MUTANT} and runs with a budget of T. For a fair comparison, EA and Mutant-UCB use the same mutation operators. We set $K_{\text{EA}} << K_{\text{MUTANT}} \lesssim K_{\text{RS}}$. Indeed, as each configuration is fully trained in the evolutionary algorithm, K_{EA} must be much lower than T/N to allow the creation of a sufficient number of offsprings. Similarly, Mutant-UCB mutation operator will create new configurations during the optimization procedure. However the evolutionary algorithm will generate even more individuals with the crossover, so K_{MUTANT} may be higher than K_{EA}. We then set K_{MUTANT} a bit smaller than T/N. We emphasize that, with the creation of offsprings and mutants, the final number of evaluated models by the evolutionary algorithm and Mutant-UCB will be much higher than K_{EA} and K_{MUTANT}, respectively. In addition, Random Search and the evolutionary algorithm fully train each configuration tested (of which there are T/N), while Hyperband and Mutant-UCB allow some of them to be partially trained (resulting in a final population of more than T/N configurations).

3.4.2 Results

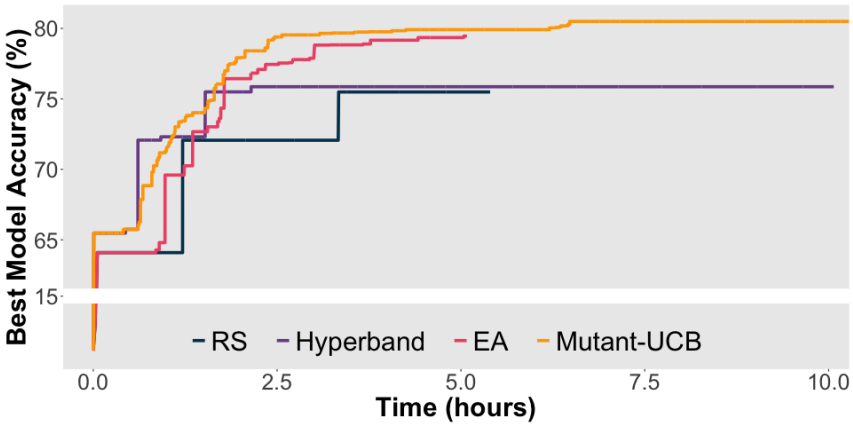
Table 3.1: Number of tested models and accuracies (in %) of the best model for Random Search (RS), asynchronous evolutionary algorithm (EA) and Mutant-UCB on CIFAR-10, MRBI and SVHN data sets.

Data set	CIFAR-10	MRBI	SVHN
RS	1 000 · 75.3	1 000 · 75.5	1 000 · 90.7
EA	1 000 · 77.1	1 000 · 79.5	1 000 · 91.9
Hyperband	2 400 · 75.4	2 400 · 75.9	2 400 · 91.0
Mutant-UCB	**3 399 · 79.5**	**3 463 · 80.5**	**3 471 · 92.4**

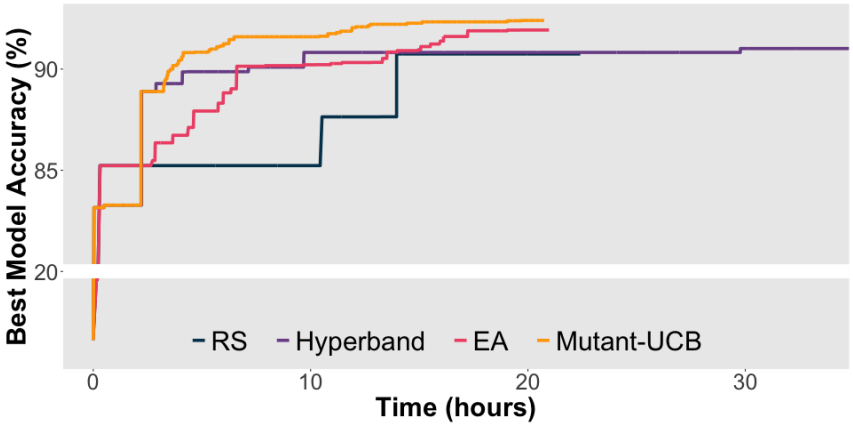
We run the experiments with $T = 10,000$, $N = 10$ and $E = 0.05$ for Mutant-UCB. The tuning of the parameter E for Mutant-UCB is not as important as for the UCB-E algorithm. We only need the algorithm to not explore too much, since the mutation



(a) CIFAR-10.



(b) MRBI.



(c) SVHN.

Figure 3.1: Accuracy of the best model over computational time for Random Search (RS), asynchronous evolutionary algorithm (EA) and Mutant-UCB on CIFAR-10, MRBI and SVHN data sets.

operator represents an additional form of exploration. Indeed, in the UCB-E algorithm, parameter E is responsible for managing the balance between exploration (through the execution of a few sub-trains for numerous models) and exploitation (through the execution of numerous sub-trains for a few models). Here, the Mutant-UCB algorithm incorporates two forms of exploration: firstly, through the sub-training of numerous models (which is also linked to the initial number of models K), and secondly, through the generation of mutants. If E is relatively large, numerous models will be tested, with a correspondingly smaller number of mutants generated (such an algorithm would be similar to Random Search). Conversely, if E is relatively small, a limited number of models will be fully trained, resulting in a substantial number of mutants being generated from them (such an algorithm would be similar to an evolutionary algorithm without the crossover operator, selecting only the best models at each iteration). A discussion on the tuning of the exploration parameter E can be found Appendix B.2.

Each sub-trains contains 10 epochs, resulting in a maximum of 100 training epochs, and the learning rate is set to 0.01. Each experiment is run on a HPC environment using 20 NVIDIA V100 GPUs.

We display Table 3.1 the maximum accuracies and the number of tested models for each algorithm from our baseline. We see that Mutant-UCB outperforms Random Search, the evolutionary algorithm and Hyperband for every data sets. The use of the mutation operator seems to be the primary factor in this performance. Indeed, the evolutionary algorithm comes well ahead of Hyperband and Random Search. It would seem that digging around promising solutions leads to better configurations. However, resources allocation also seems to be a key factor. Hyperband is in fact slightly better than Random Search, and converges much faster, as it can be seen in Figure 3.1. The computation times to perform the T iterations vary a lot between the algorithms and the task at hand. The hardware definitely has an impact on this, but the resources allocation will also play a crucial role. Throughout the paper there is an implicit assumption that a sub-train's budget in terms of memory and time is independent of the model, which is not entirely correct. Indeed, performing more sub-trains on configurations which are more complex take generally a longer time and affect the total duration of the experiment. Mutant-UCB, with both the mutation operator and the resources allocation has the fastest convergence and yields the better accuracies. Appendix B.1 details the models found by the different algorithms.

3.5 Discussion

This work presents Mutant-UCB, an innovative model selection algorithm, which combines a UCB-based bandit algorithm with evolutionary algorithm operators. Most configuration selection approaches, such as Bayesian optimization or continuous bandit algorithms, typically consider a normed vector space to represent the pool of possible configurations. These approaches assume that the reward function is smooth, meaning that

two configurations that are close in the underlying vector space lead to close accuracies. Mutant-UCB and the other algorithms in the baseline do not require any smoothness assumptions. Besides, thanks to its resource allocation, Mutant-UCB demonstrates a high exploratory potential. It can evaluate more models within a similar budget compared to Random Search or evolutionary algorithms. For example, on the MRBI data set, with a budget $T = 10,000$, Mutant-UCB evaluated $3,500$ configurations, while the Evolutionary Algorithm and Random Search only evaluated $1,000$. The use of a mutation operator, on the other hand, reinforces the exploitation of promising solutions and allows us to reach much higher performance configurations than Hyperband and Random Search. The mutation can be viewed as a concept of proximity: a mutant and its original model are close together, as defined by the chosen operator (which does not require any normed vector space). It remains more permissive than the operators of the evolutionary algorithm. In particular, the crossover from the evolutionary algorithm requires homogeneity between the elements of the search space, unlike Mutant-UCB. Thus, Mutant-UCB could be used with a search space that combines various machine learning models, such as neural networks, random forests, or boosting; as soon as we define a mutation operator for each type of model. Finally, the Mutant-UCB algorithm is highly scalable in an HPC environment because configurations are evaluated independently and asynchronously, in contrast to Hyperband and classical evolutionary algorithms, which evaluate populations synchronously. In summary, Mutant-UCB has several advantages that make it an attractive algorithm, in addition to its baseline-beating performance demonstrated in the previous section. One disadvantage is the need to store the weights of all previous configurations evaluated. This is because the pool of solutions on which we apply the UCB part of the algorithm is not limited, unlike the other algorithms from the baseline.

Prospects. The experiments demonstrate the relevance of Mutant-UCB. A challenge for further work would be to obtain an upper bound on the simple regret of this algorithm to attest a theoretical performance. In this best-arm identification problem for infinite-armed bandit framework; the simple regret will be the accuracy of the best possible model minus the accuracy of the model selected by the algorithm. To obtain theoretical results, we believe that it will be necessary to introduce some concepts from sleeping bandits due to the creation of mutants (see, e.g., Kleinberg et al. 2010 and contextual bandits to model the proximity between mutants and their original models (see, among other Li et al. 2010). One of the most challenging aspects of the analysis will be to select an appropriate hypothesis regarding the distribution of rewards conditionally to the chosen arm. The classical stochastic bandit assumptions are not applicable in this context, as they suggest that, conditionally to the chosen arm, the accuracy does not depend on the number of sub-trains performed. However, empirical evidence indicates that performing multiple sub-trains enhance performance. Furthermore, in order to legitimize the idea of integrating mutation operations, and hopefully get the simple regret bound, it seems

essential to add an assumption about the distribution of the accuracies of mutants. It should also be noted that the HPC environment and the neural network training duration puts us in a context where rewards arrive with a delay (namely, in a delayed bandits framework - see, e.g., Vernade et al. 2020) and not all arms are available at all times (sleeping bandits again).

The strategy for creating mutant models is independent of the choice of bandit algorithm used to select the arms. Alternative approaches, other than those based on UCB, could be considered.

In this chapter we applied Mutant-UCB to a very generic problem: neural networks optimization for image classification. The flexibility of this algorithm means that it can be applied to a wide range of problems. A natural extension of this work would be to apply Mutant-UCB to a variety of tasks, models and search spaces where state-of-the-art algorithms, by their very nature, would be limited or even unusable.

Chapter 4

DRAGON: a Python package for Automated Deep Learning

In this chapter, we propose a tutorial on the Python package DRAGON, which implements the DAG-based search space and the various search algorithms mentioned in the previous two chapters (including the Evolutionary Algorithm and Mutant-UCB). DRAGON is presented as an automated deep learning toolbox that offers different levels of customization to create task-specific AutoDL frameworks for any given task. The package is organized around three main concepts: the search space, the search operators, and the search algorithms. In the following, we will explain how they are implemented and how they can be used. We conclude with a more applied section that gives a more concrete example of how the package can be used. This chapter is not intended to be an exhaustive presentation of the package. More information, especially about the implementation, can be found in the documentation[1].

Contents

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

4.1 Introduction

DRAGON, for DiRected Acyclic Graphs optimization, is an open source Python package designed to optimize the hyperparameters and architectures of deep neural networks. It implements the algorithmic framework proposed Chapter 2. In this framework, Deep Neural Networks are encoded as Directed Acyclic Graphs (DAGs), where the nodes can be any *Pytorch* operations parameterized by some optimizable hyperparameters and the edges are the connections between them. While the framework is already presented as more flexible than the AutoDL approaches in the literature (e.g., the possibility to choose candidate operations, optimization of hyperparameters, and no constraints on connections), its implementation, called DRAGON, offers even more choices to the user. This chapter provides a detailed overview of the package and illustrates its customization options. Unlike most AutoDL or AutoML packages in the literature such as AutoGluon (Erickson et al., 2020), Auto-Keras (Jin et al., 2019) or AutoPytorch (Zimmer et al., 2021), DRAGON is not a no-code package. It is not enough to write `.fit` and then `.predict` to get results with DRAGON. To use the package, the user must define an appropriate search space, a meta-architecture, and procedures for training and validation. Although this implementation requires more initial effort, it allows for a wide range of tools tailored to different problems. The search space consists of objects that fall into three main categories: search space variables, search operators, and search algorithms.

The search space variables are *Python* objects able to encode various elements such as integers or arrays. These variables can be combined to represent more complex objects such as neural network layers or DAGs. While a variety of encodings are already available in the package, we will detail how to create custom variables to further extend the package with other representations.

To navigate within the search space and modify the configurations within it, mutation or neighborhood operators are defined for each variable. The idea is to slightly modify a representation to obtain one of its neighbors. The assumption is that if the original configuration gave good results, then there must be other configurations in its neighborhood that perform equally well or better. As with variables, the package comes with ready-made functions, but these operators can also be customized.

These search operators can be used by search algorithms. To date, four algorithms have been implemented in DRAGON: a Random Search, HyperBand (Li et al., 2018), the Evolutionary Algorithm, and Mutant-UCB respectively introduced Chapters 2 and 3. The former two do not need to change the configurations and therefore do not need mutation operators while the latter do use them. These four algorithms depend on the same Python class, which is described in detail in this tutorial. It manages the storage of evaluated configurations and the potential distribution of the algorithm in a High-Performance Computing (HPC) environment. This implementation makes it easy to add new search algorithms without having to manage the hardware.

Finally, the part not covered by DRAGON at all, the performance evaluation, is presented in the form of examples. It consists of creating a neural network from elements in the search space, training it, and validating it.

4.2 Search Space variables

The search space design is based on an abstract class called `Variable`, originally proposed within a hyperparameters optimization package called zellij[2]. This class can represent various objects from float to tree-based or array-like structures. Each child class should implement a `random` method, which represents the rules defining the variable and how to generate random values. The other method to implement is called `isconstant` and specifies whether the variable is a constant or not. Variables can be composed to represent more or less complex objects. Among the composed variables, some have been created specifically for the DAG encodings. A variable can be extended by `Addons` to implement additional features such as the search operators detailed in Section 4.3. The structure of a `Variable` definition is the following:

```
from dragon.search_space.base_variables import Variable

class CustomVar(Variable):

    def __init__(self, label, **kwargs):
        super(CustomVar, self).__init__(label, **kwargs)
        """
        The label is the unique identifier of a variable.
        """

      def random(self, size=None):
          """
          Create ‘size‘ random values of the variables
          """

      def isconstant(self):
          """
```

[2] https://zellij.readthedocs.io/en/latest/

```
            Specify is a variable is a constant or not.
            This function might depends on the variable attributes
.
            """
```

All the variables ihnerit from the abstract class `Variable`. An example of the implementation of a variable for an integer can be found below:

```
    class IntVar(Variable):
        """
        IntVar
        Defines a Variable discribing integer variables.

        Parameters
        ----------
        label : string
            Label of the variable.
        lower : int
            Lower bound of the variable.
        upper : int
            Upper bound of the variable
        """
        def __init__(
          self, label, lower, upper, **kwargs):
          super(IntVar, self).__init__(label, **kwargs)
          self.low_bound = lower
          self.up_bound = upper + 1

        def random(self, size=None):
          """
          `size` integers are randomly drawn form the interval `[
low_bound, up_bound]`.
          """
          return np.random.randint(self.low_bound, self.up_bound,
size, dtype=int)

        def isconstant(self):
          """
          An IntVar is a constant if the upper and the lower bounds
 are equals.
          """
          return self.up_bound == self.low_bound
```

4.2.1 Base variables

The base variables implement basic objects such as integers, floats or categorical variables. Each of them is associated with a `Variable`, defining what values an object can take. For example, an integer object is implemented by the example variable `IntVar`

suggested above. It takes as arguments the lower and upper bounds defining the range of values the integer might take.

```
from dragon.search_space.base_variables import IntVar

v = IntVar("An integer variable", 0, 5)

v.random()
3
```

In this example, the variable `v` defines an integer taking values from 0 to 5. When calling `v.random()`, the script returns an integer from this range, here 3. The base variables available in DRAGON are listed in Table 4.1. Note that the features from a `CatVar` variable might include objects from the class `Variable` and non-variables objects.

Table 4.1: Base variables

Type	**Variable Name**	**Main Parameters**
Integer	`IntVar`	Lower / upper bound
Float	`FloatVar`	Lower / upper bound
Categorical (string, etc)	`CatVar`	Features: list of possible choices
Constant (any object)	`Constant`	Value: constant value

4.2.2 Composed variables

The base variables can be composed to create more complex objects such as arrays of variables.

```
from dragon.search_space.base_variables import *

a = ArrayVar(
    IntVar("int1",0,8),
    IntVar("int2",4,45),
    FloatVar("float1",2,12),
    CatVar("cat1", ["Hello", 87, 2.56])
    )

a.random()
[5, 15, 8.483221226216427, 'Hello']
```

Here we have created an array of four different elements: two integers, one between 0 and 8 and the other between 4 and 45, a float between 2 and 12, and a categorical variable that takes values between `["Hello", 87, 2.56]`. For example, an `ArrayVar` can represent a list of hyperparameters of a machine learning model. Unlike the `CatVar` features attribute that may contain `Variable` and non-`Variable` elements, the attributes of composed variables must inherit from the class `Variable`. This means that we cannot

create an `ArrayVar` with a simple 5 as argument. To include a constant integer, we have to encode it using a `Constant` variable.

Table 4.2: Composed variables

Definition	Variable Name	Main Parameters
Array of Variables	`ArrayVar`	List of `Variable`
Repeated `Variable`	`Block`	`Variable` that will be repeated and the number of repetitions.
Random number of repetitions	`DynamicBlock`	`Variable` that will be repeated and the maximum number of repetitions.

4.2.3 Deep Neural Networks Encoding

Both base and composed variables have been used to encode Deep Neural Networks architecture and hyperparameters.

4.2.3.1 Operations and hyperparameters encoding

Deep Neural Networks are made of layers. In DRAGON's case, those layers are `nn.Module` from *PyTorch*. The user can integrate any base or custom `nn.Module`, but has to wrap it into a `Brick` object. This *Python* class takes an input shape and some hyperparameters as arguments and initializes a given `nn.Module` with these hyperparameters so it can process a tensor of the specified input shape. The forward pass of a `Brick` can directly apply the layer to an input tensor, or it can be more complex and transform the input data before the operation. Finally, the abstract class `Brick` also implements a `modify_operation` method. It takes an `input_shape` and modifies the shape of the operation weights so that it can take as input a tensor of shape `input_shape`. This method is applied when the Deep Neural Network is created or modified.

```
import torch.nn as nn
from dragon.search_space.cells import Brick

class Dropout(Brick):
    """
    Dropout
    Implementation of a dropout layer as a DRAGON candidate
   operation.

    Parameters
    ----------
    input_shape : tuple
        Shape of the incoming vector.
    rate : float
```

```
        Probability of an element to be zeroed.
    """
    def __init__(self, input_shape, rate):
        super(Dropout, self).__init__(input_shape)
        self.dropout = nn.Dropout(p=rate)
    def forward(self, X):
        X = self.dropout(X)
        return X
    def modify_operation(self, input_shape):
        pass
```

```
from dragon.search_space.cells import Brick
import torch.nn as nn

class MLP(Brick):
    """
    MLP
    Implementation of a linear layer as a DRAGON candidate
   operation.

    Parameters
    ----------
    input_shape : tuple
        Shape of the incoming vector.
    out_channels : int
        Size of each output sample.
    """
    def __init__(self, input_shape, out_channels):
        super(MLP, self).__init__(input_shape)
        self.in_channels = input_shape[-1]
        self.linear = nn.Linear(self.in_channels, out_channels)
    def forward(self, X):
        X = self.linear(X)
        return X
    def modify_operation(self, input_shape):
        d_in = input_shape[-1]
        diff = d_in - self.in_channels
        sign = diff / abs(diff) if diff != 0 else 1
        pad = (int(sign * ceil(abs(diff)/2)),
                int(sign * floor(abs(diff)/2)))
        self.in_channels = d_in
        self.linear.weight.data =
            nn.functional.pad(self.linear.weight, pad)
```

The two blocks of code above show the implementation of a `Dropout` layer and an `MLP` (Multi-Layer Perceptron), respectively. While the wrapping of the `Dropout` layer into a `Brick` object requires minimal modifications, the `MLP` wrapping necessitates some effort to implement the `modify_operation` method. Indeed, the shape of the weights of an `nn.Linear` operation depends on the input tensor dimension.

The variable encoding a Brick is called HpVar. It takes as input a Constant or a CatVar containing a single or several layers implemented a Bricks, as well as a dictionary of hyperparameters. If a CatVar is given as input operation, all the Bricks contained in the CatVar's features should share the same hyperparameters.

```
from dragon.search_space.bricks import MLP
from dragon.search_space.base_variables import Constant, IntVar
from dragon.search_space.dag_variables import HpVar

mlp = Constant("MLP operation", MLP)
hp = {"out_channels": IntVar("out_channels", 1, 10)}
mlp_var = HpVar("MLP var", mlp, hyperparameters=hp)
mlp_var.random()
[<class 'dragon.search_space.bricks.basics.MLP'>,
   {'out_channels': 9}]
```

```
from dragon.search_space.bricks import LayerNorm1d, BatchNorm1d
from dragon.search_space.base_variables import CatVar
from dragon.search_space.dag_variables import HpVar

norm = CatVar("1d norm layers", features=[LayerNorm1d, BatchNorm1d
   ])
norm_var = HpVar("Norm var", norm, hyperparameters={})
norm_var.random()
[<class 'dragon.search_space.bricks.normalization.BatchNorm1d'>,
   {}]
```

These two examples show how to use HpVar with a Constant and a CatVar operation respectively. Here, the CatVar is made here of two versions of normalization layers which share the same hyperparameters (none in this example). The only hyperparameter that can be optimized for the MLP layer is the size of the output channel, here, an integer between 1 and 10. To facilitate the use of DRAGON, operations (such as convolution, attention, pooling, or identity layers) as Brick and their variable HpVar are already implemented in the package.

4.2.3.2 Node encoding

DRAGON implements Deep Neural Networks as computational graphs, where each node is a succession of a combiner, an operation and an activation function, as detailed in Chapter 2. The operation is encoded as a Brick, as mentioned above. The combiner unifies the (potential) multiple inputs the node might have into one unique tensor. The combiners available in DRAGON are add, mul and concat and are encoded as strings. The activation function can be any *PyTorch* implemented or custom activation function. An nn.Module object called Node takes as inputs these three elements to create a node. A Node implements several methods. The main ones are:

- set_operation: takes as input a variable input_shapes containing the input shapes of the incoming tensors. The method uses the combiner to compute the

operation input shape and initialize the operation weights with the right shape. The initialized operation is then used to compute the node output shape. This value will be used by the `set_operation` methods from the child nodes of the current one.

- `modification`: modifies the input shapes, the combiner, the operation or the hyperparameters of the node. The modification can happen after a mutation or a change in the input tensor shape. If the operation is not modified, the method `modify_operation` from the `Brick` object is called only to modify the weights without creating a new operation. This way, weights' optimization does not restart from the beginning.
- `set`: automatically chooses between the `set_operation` and `modification` methods depending on the context.
- `forward`: computes the node forward pass from the combiner to the activation function.

The `Variable` corresponding to a `Node` is called `NodeVariable`. It takes as input a `Constant` or a `CatVar` for the combiner and the activation functions. The operation is implemented as an `HpVar` as mentioned above. However, a node can have multiple candidate operations, all of them implemented as different `HpVar` objects. In this case, instead of directly being given as an `HpVar`, they are contained within a `CatVar`. The `CatVar` features will contain the different `HpVar`. An example is given below.

```
from dragon.search_space.base_variables import CatVar
from dragon.search_space.bricks_variables import activation_var

# Operation encoded as a `CatVar` of `HpVar`
operations = CatVar("Candidates", [mlp_var, norm_var])
candidates = NodeVariable(
    label="Candidates",
    combiner=CatVar("Combiner", features=["add", "concat"]),
    operation=operations,
    activation_function=activation_var("Activation")
)
```

The activation functions are encoded through the `activation_var` object. It is a default `CatVar` implemented within DRAGON which contains the basic activation functions from *PyTorch*. The `random` method from the `NodeVariable` randomly selects a combiner, an activation function, an operation (in case of a `CatVar` operation), and draws random hyperparameters.

4.2.3.3 DAG encoding

Finally, the last structure presented is the DAG, which (partially) encodes a Deep Neural Network. The object that encodes the graphs is called `AdjMatrix` and is

also a nn.Module. It takes as arguments a list of nodes and an adjacency matrix (a two-dimensional array) representing the edges between these nodes. A method assert_adj_matrix is used to evaluate the correct format of the adjacency matrix (e.g, right number of rows and columns, upper triangular, diagonal full of zeros). The directed acyclic structure of the graph allows ordering the nodes as explained in Chapter 2. Just like the Node object, the AdjMatrix implements a method set that takes as an argument input_shape and calls the method set from each node in that order. The forward pass computation is also done in this order. During the forward computation, the outputs are stored in a list to be used for later nodes of the graph that have them as input.

The Variable that represents the AdjMatrix is called EvoDagVariable. It takes as input a DynamicBlock whose repeated variable would be a NodeVariable. This NodeVariable will have its operation encoded as CatVar in case of multiple candidate layers. A random AdjMatrix is created by first drawing the number of nodes from the graph. Then a random value of NodeVariable is drawn for each node. Finally an AdjMatrix of the right dimension is created.

Figure 4.1 below illustrates how the elements are linked together. The hierarchical composition of the variables creating a DAG allows optimization at various levels, from the graph structure to any operation hyperparameters. Hyperparameters or operations can be imposed to reduce the search space by passing certain operations with constant hyperparameters Constant. This way, we can reconstruct more constrained search spaces close to cell-based search spaces.

4.3 Search Operators

Once the search space is defined, a simple random search can be used to find good configurations. However, training a deep neural network is a long and resource-intensive process. Therefore, the search algorithm must train neural networks as little as possible. It is therefore essential to use information from previous training and evaluation. By identifying good solutions, they can be modified to optimize their performance. To make these modifications, a neighbor attribute can be associated with each of the variables defined in Section 4.2. They can be thought of as neighborhood or mutation operators. These attributes are added using Addons. A Addon is an object that is linked to another. It allows extending functionalities of the other object (in this case a Variable) without modifying its implementation. DRAGON provides an implementation of one neighborhood per variable, but users can implement their own. A neighborhood is a class inheriting from the abstract class VarNeighborhood, which is an Addon. It should have the following structure:

```
1 class CustomInterval(VarNeighborhood):
2     """
3     CustomInterval
```

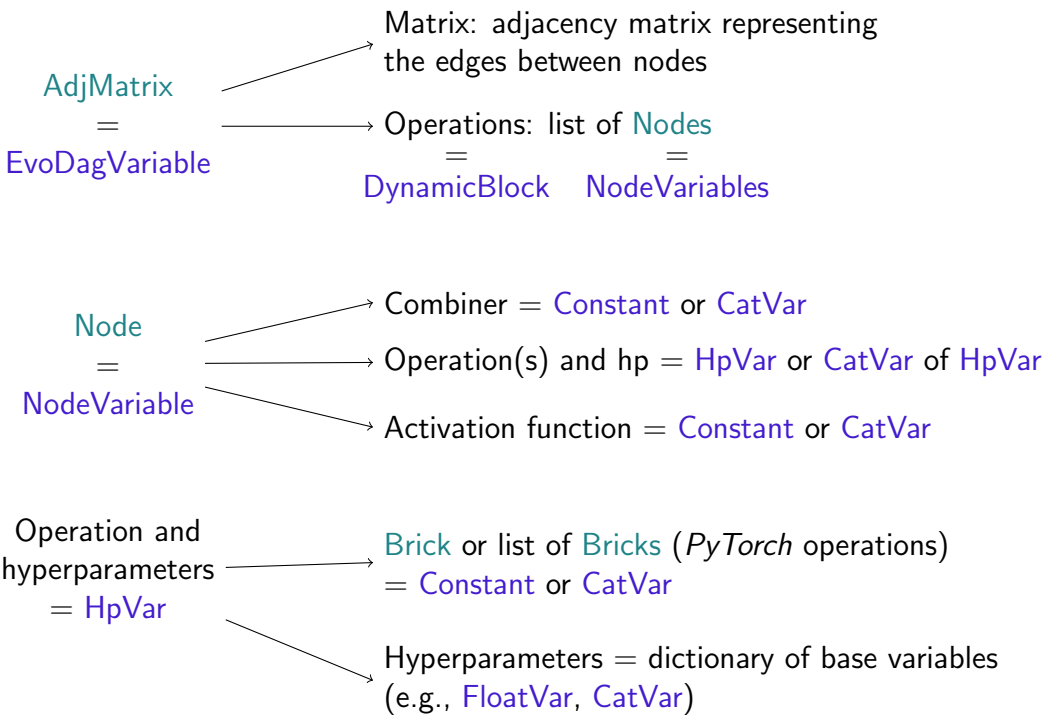


Figure 4.1: Summary of DAG Encoding within DRAGON

```
    Addon used to determine the neighbor function of a Variable.

    Parameters
    ----------
    variable : Variable, default=None
        Targeted Variable.
    neighborhood : any, default=None
        Parameter of the neighborhood
    """
    def __call__(self, value, size=1):
        """
        Function defining how to choose `size` neighbors surrounding the variable `value`.
        """

    @VarNeighborhood.neighborhood.setter
    def neighborhood(self, neighborhood):
        """
        Set the neighborhood parameter (e.g., probability to mutate ArrayVar values).
        """
        self._neighborhood = neighborhood

    @VarNeighborhood.target.setter
    def target(self, variable):
```

```
        """
        Provide the CustomInterval object with information about
    the related `Variable`.
        """
        self._target = variable
```

Here is an example of a neighborhood defined for the `IntVar` variable. The neighborhood selects new values within an interval surrounding the current one, parameterized by an interval size:

```
from dragon.search_space.addons import VarNeighborhood

class IntInterval(VarNeighborhood):
    def __call__(self, value, size=1):
        """
        Get the upper bound for the interval:
        the minimum between [current value + neighborhood] and
        [the maximum value the variable can take].
        """
        upper = min(value + self.neighborhood + 1, self.target.
    up_bound)

        """
        Get the lower bound for the interval:
        the maximum between [current value - neighborhood] and
        [the minimum value the variable can take].
        """
        lower = max(value - self.neighborhood, self.target.
    low_bound)

        res = []
        for _ in range(size):
            v = np.random.randint(lower, upper)
            while v == value:
                v = np.random.randint(lower, upper)
            res.append(int(v))

        return res if size > 1 else res[0]

    @VarNeighborhood.neighborhood.setter
    def neighborhood(self, neighborhood):
        self._neighborhood = neighborhood

    @VarNeighborhood.target.setter
    def target(self, variable):
        self._target = variable
```

This `IntInterval` is assigned to the `Variable` while we define it:

```
from dragon.search_space.base_variables import IntVar

```

```
v = IntVar("An integer variable", 0, 5, neighbor=IntInterval(
    neighborhood=1))
v.neighbor(4)  # Example usage
# Output: 3
```

4.3.1 Base and composed variables

The neighborhood operators available for base and composed variables within DRAGON are listed in Table 4.3. If a composed variable has a `neighbor` addon, then all the values

Table 4.3: Base and composed neighborhoods

Type	Variable Name	Neighbor Name	Main Parameters
Integer	`IntVar`	`IntInterval`	Interval size
Float	`FloatVar`	`FloatInterval`	Interval size
Categorical (string, etc)	`CatVar`	`CatInterval`	
Constant (any object)	`Constant`	`ConstantInterval`	
Array of Variables	`ArrayVar`	`ArrayInterval`	
Fix number of repeats	`Block`	`BlockInterval`	
Random number of repeats	`DynamicBlock`	`DynamicBlockInterval`	Neighborhood of the `DynamicBlock` size

composing this variable should have a `neighbor` addon. For example with a `Block`:

```
    from dragon.search_space.base_variables import Block, FloatVar
    from dragon.search_algorithm.base_neighborhoods import
BlockInterval, FloatInterval

    content = FloatVar("Float example", 0, 10, neighbor=
FloatInterval(2))
    a = Block("Block example", content, 3, neighbor=BlockInterval()
)

    a.neighbor([2, 1, 6])
    [2, 1.3432682541165653, 7.886611679292923]
```

In this example, the `Block`'s value is a `FloatVar` variable. `Neighbor` addons are given to both `Variables`. The addon `BlockInterval` makes use of the `FloatInterval` to create the new value.

4.3.2 DAG encoding neighborhoods

Besides the base and composed variables, the ones used for DAG encoding, namely `HpVar`, `NodeVariable` and `EvoDagVariable` also have implemented neighborhoods.

4.3.2.1 Operation neighborhood

The HpVar neighborhood is called HpInterval. Its arguments are an operation and a set of hyperparameters. It selects among the operation and the various hyperparameters the ones that will be mutated. The mutation applied to the operation is not more likely to be called than the hyperparameters one. It does not have any effect if the operation is a Constant. The chosen hyperparameters are mutated according to their neighbor addon. The HpInterval object returns the new operation and hyperparameters. It is possible to modify this operator to increase the probability of modifying the operation or to prevent hyperparameter mutations before a certain iteration of the search algorithm.

4.3.2.2 Node neighborhood

If the operation within a NodeVariable is encoded as a HpVar, then its neighborhood will be the HpInterval. But, in the case of CatVar of HpVar, when dealing with candidate operations implemented in various HpVar, the neighborhood for the operation is called CatHpInterval. This neighborhood chooses between modifying the current operation or drawing a completely new one. It takes as argument a probability $p \in [0, 1]$ of only modifying the current operation (by default equal to 0.9). With a probability p, the function will look for the HpVar corresponding to the current value and call the HpInterval of this variable. The matching is done by looking at the features attribute if the HpVar operation is a CatVar or the value attribute if the operation is a Constant. With a probability $1 - p$, a new layer is drawn (with a new operation and new hyperparameters), by calling the random function of the CatVar.

The neighborhood class associated with a NodeVariable is called NodeInterval. It selects among the combiner, the operation and the activation function what is to be modified. For the selected elements, their neighbor attributes are invoked. In the current implementation, the chances for modifying any of these three elements are the same, which may be changed.

4.3.2.3 DAG neighborhood

The EvoDagVariable neighborhood class is called EvoDagInterval. This neighborhood may perform five types of mutations, mentioned Chapter 2:

- Adding a node
- Deleting a node
- Modifying a node
- Modifying the input connections of a node
- Modifying the output connections of a node

First, it randomly selects the nodes to be modified. A parameter `nb_mutations` can be set to limit the number of nodes modified. For each selected node, the allowed mutations can be different. For example, if the selected node is the last one, its outgoing connections cannot be changed. If the maximum number of nodes is reached, the `add` mutation cannot be used. For each node, once the set of allowed mutations has been defined, one value from that set is drawn and performed. After a mutation, some tests are performed to ensure a correct adjacency matrix structure and to adjust connections if necessary. This prevents nodes from having no incoming or outgoing connections. By modifying the nodes and edges, the input tensors of each node may have changed shape. In this case, the node's operation is modified by calling its `modification` to adjust the weights.

4.3.3 Crossover

Besides the neighborhood operators, a crossover has been implemented to use DRAGON with an evolutionary algorithm. The crossover is not an `Addon`, it is a simple class implementing a two-point crossover. The crossover `__call__` method takes as input two individuals `ind1` and `ind2` which should be array-like variables, with the same types of `Variables` at each position. Two index points from arrays are picked randomly. The segment between those two index points is swapped between the parents. For each element of this segment, if one of them is an `AdjMatrix` variable, then the DAG-based crossover is used. The DAG-based crossover takes as input two `AdjMatrix` elements and perform the following operations:

- Selects the indexes of the operations that would be exchanged in each graph.
- Removes the corresponding lines and columns from both adjacency matrices.
- Computes for each exchanging node, the index in the other graph where it will be inserted.
- Inserts the new rows and columns within both adjacency matrices.
- Asserts no nodes without incoming or outgoing connections are remaining within the matrices.
- Asserts the new matrices are upper-triangular.
- Creates new `AdjMatrix` variables with the new nodes and matrices.

The DAG-based crossover is presented in more detail Chapter 2.

4.4 Search Algorithms

4.4.1 Main structure

The variables and their operators respectively defined in Sections 4.2 and 4.3 can be used to implement various search algorithms. DRAGON uses an abstract class called `SearchAlgorithm` to structure the algorithms.

```
class SearchAlgorithm:
    """SearchAlgorithm
        Abstract class describing the general structure of a search algorithm.
        The classes inheriting from the `SearchAlgorithm` abstract class should implement a `select_next_configuration` and a `process_evaluated_configuration` methods.

        Parameters
        ----------
        search_space: `Variable`
            `Variable` containing all the design choices from the search space. It should implement a `random` method and a `neighbor` one if necessary.
        n_iterations: int
            Number of iterations.
        population_size: int
            Size of the randomly initialized population.
        evaluation: function
            Performance evaluation function. Takes as argument a set of configuration and the unique index of this configuration. Returns the performance and the model built.
    """
    def __init__(self, search_space, n_iterations, population_size, evaluation):
        self.search_space = search_space
        self.n_iterations = n_iterations
        self.population_size = population_size
        self.evaluation = evaluation
        self.min_loss = np.inf

    def run(self):
        # Generate the first population of configurations
        self.population = self.create_first_population()

        # Evaluate and process each of these configurations
        for i, p in enumerate(self.population):
            loss = self.evaluation(idx)
            self.process_evaluated_configuration(idx, loss)
            if loss < self.min_loss:
                self.min_loss = loss
```

```
34
35      t = population_size
36      # While the number of iterations has not been reached
37      while t < n_iterations:
38          # Select the next configurations
39          idx_list = self.select_next_configurations()
40          for idx in idx_list:
41              # Evaluate the configurations
42              loss = self.evaluation(idx)
43              # Process the evaluated configuration
44              self.process_evaluated_configuration(idx, loss)
45              if loss < self.min_loss:
46                  self.min_loss = loss
47              t += 1
```

Listing 4.1: Example of a search algorithm

This pseudo-code is a highly simplified, schematic version of the `SearchAlgorithm` class to help illustrate its main aspects. The class input arguments depend on the application. The search space is a (composed) `Variable` that can represent all considered configurations. The `evaluation` function takes a configuration as input, builds the model, trains and evaluates it, and then returns a `loss` value representing its performance. This function depends on the tasks at hand and should be implemented by the user. It is important to note that the class `SearchAlgorithm` is made to minimize a value, and thus the `evaluation` function should return a loss, not a reward. The number `n_iterations` represents the amount of calls to the `evaluation` function during training. The `SearchAlgorithm` class assumes that all search algorithms go through an initialization phase where a certain population is randomly generated and evaluated. Then, until a maximum number of iterations is reached, a configuration is selected and evaluated by a `select_next_configuration` function, specific to the algorithm in question. Depending on the performance obtained, the algorithm will process this configuration with the `process_evaluated_configuration` function. A pseudo-code of a `select_next_configuration` function for the Evolutionary Algorithm is given below.

```
1 def select_next_configurations(self):
2     parent_1, parent_2 = tournament_selection(self.population)
3     offspring_1, offspring_2 = crossover(parent_1, parent_2)
4     offspring_1 = self.search_space.neighbor(offspring_1)
5     offspring_2 = self.search_space.neighbor(offspring_2)
6     return [offspring_1, offspring_2]
```

Listing 4.2: Evolutionary Algorithm selection of next configurations

It first selects two parent configurations using a tournament selection strategy. The parents are then modified using a crossover and the mutations implemented as `neighbor` attributes. When implementing a new search algorithm using the `SearchAlgorithm` structure, this function has to be specified.

4.4.2 Storage management for Deep Neural Networks

The objects encoding DAG such as the `Bricks`, `Nodes`, and `AdjMatrix` are all `nn.Module` that contain trained weights. Therefore, they can take up a lot of memory. To prevent the system from handling too many neural networks, the configurations are cached while the search algorithm is running and only loaded for evaluation or to create a new configuration by mutating them. For this purpose, each configuration is assigned a unique number called `idx`. A `storage` dictionary summarizes all the information needed by the search algorithm for each configuration. All algorithms require the `loss` for each configuration. However, some algorithms leveraging resource allocation may need additional information such as the number of resources a configuration has already received. The `storage` dictionary is updated during the `process_evaluated_configuration`. A pseudo-code of the one used by Mutant-UCB is given below.

```
def process_evaluated_configuration(self, idx, loss):
    # If the configuration has already been evaluated
    if idx in self.storage.keys():
        self.storage[idx]['Loss'] = loss
        self.storage[idx]['UCBLoss'] = (loss +
            self.storage[idx]['N_bar']*
            self.storage[idx]['UCBLoss'])/
            (self.storage[idx]['N_bar']+1)
        # Count number of time the configuration has been picked
        self.storage[idx]['N'] +=1
        # Count number of time the configuration has been trained
        self.storage[idx]['N_bar'] +=1
    else:
        self.storage[idx] =
            {"N": 1, "N_bar": 1, "UCBLoss": loss, "Loss": loss}
```

Listing 4.3: Mutant-UCB configuration processing function

This function should also be specified when implementing a new search algorithm. The file-based backup system also makes it easy to resume an aborted optimization. The `SearchAlgorithm` class incrementally saves a CSV file containing information about previously evaluated configurations. This file is used to find the stage where the search algorithm stopped and to continue the search. This is done using the `recover_optimization` method of the class.

4.4.3 Distributed version through MPI

The class `SearchAlgorithm` allows the distribution of the algorithms on multiple computation nodes of a High Performance Computing (HPC) architecture. This is based on a Message Passing Interface (MPI) with the mpi4py package[3]. The search algorithm relies on a master process and several workers processes, each assigned to a GPU. The

[3] https://mpi4py.readthedocs.io/en/stable/

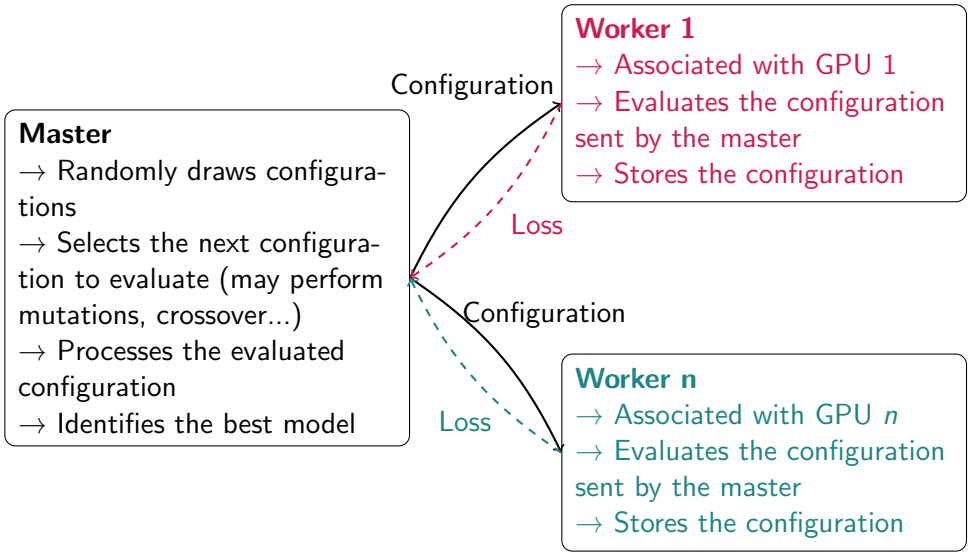


Figure 4.2: MPI implementation of the search algorithms with curved arrows.

master process performs the algorithm's main steps such as creating the population or selecting the next configurations. It dynamically sends configurations to the workers to evaluate. As soon as a process finishes an evaluation, the master processes the returned model and selects the next one to send to the worker. The worker processes can be associated with a unique GPU. They perform the training and evaluation of the configuration sent by the master. An illustration of the implementation can be found below. As training a neural network is the most time-consuming part of the search algorithm, distributing the part to several devices makes DRAGON search algorithms more efficient and easily scalable on HPC infrastructures. The `SearchAlgorithm` class activates by itself the MPI version by looking if the package mpi4py is available.

4.4.4 Implemented Algorithms

It is possible to implement new search algorithms that extend or not the `SearchAlgorithm` class. However, some are already available within the package and ready to be used. A Random Search, HyperBand (Li et al., 2018), an Evolutionary Algorithm, and, Mutant-UCB are implemented. For a given application they require the implementation of a search space, a performance evaluation function and the setting of some required parameters such as the path to save the configurations, or the number of iterations.

```
from dragon.search_algorithm.mutant_ucb import Mutant_UCB

# Initialize the search algorithm with required arguments
search_algorithm = Mutant_UCB(search_space,
    save_dir="save/test_mutant", T=20, N=5, K=5, E=0.01,
    evaluation=loss_function)
# Launch the optimization
```

```
    search_algorithm.run()
```

Listing 4.4: Example use of Mutant-UCB

4.5 Performance evaluation

DRAGON can be used for a wide range of applications. To apply the package to a particular task, one needs to:

- Create an appropriate search space with the variables introduced in Section 4.2.
- Select (or implement) a search algorithm such as those presented in Section 4.4. Depending on the type of search algorithm, it will be necessary to associate `neighbor` attributes to the variables in the search space, as described in Section 4.3.
- Implement a performance evaluation function.

This function should take as input a configuration (and the index of that configuration in the case of search algorithms implemented in the package), and return a `loss` corresponding to the evaluation of the configuration. The `loss` and the way it is calculated depends on the task at hand. In general, it is necessary to build a neural network from the configuration, train it and validate it on data sets. Let's take the example of a problem classifying a vector $X \in \mathbb{R}^n$, with $n \in \mathbb{N}^\star$, into 10 different classes. We want to find the best model among any type of architecture that takes an input of size n and outputs a vector of 10 values. This type of architecture can be represented by a DAG. However, to constrain the output to be of dimension 10, a final layer converting the output tensor of any size from the DAG to a vector of size 10 is necessary. The DAG and the output layer are associated together within a `nn.Module` called `MetaArchi` in the following example.

```
class MetaArchi(nn.Module):
    """MetaArchi
        Meta-architecture of the neural networks which should be
    optimized.

        Parameters
        ----------
        args : dict
            Dictionary containing a solution from the search space.
        input_shape : tuple
            Shape of the input vector.
    """
    def __init__(self, args, input_shape):
       super().__init__()
       # Number of features, here equals to n
```

```
        self.input_shape = input_shape


        # The Neural Network will be optimized through a DAG
        self.dag = args['Dag']
        self.dag.set(input_shape)

        # We set the final layer
        self.output = args["Out"]
        self.output.set(self.dag.output_shape)

    def forward(self, X):
        out = self.dag(X)
        return self.output(out)
```

Listing 4.5: Meta Architecture class definition

The class `MetaArchi` takes as arguments the variable `args` which contain the configuration indicating how to build the model and `input_shape` indicating the shape of the tensors that will be processed. The argument `self.dag` is an `AdjMatrix` and `self.output` is a `Node`. Their methods `set` were explained Section 4.2. They adjust the `nn.Module` weights to ensure they can handle tensors with the right input shape. The search space used to create the variable `args` is specified by the user. In the following example, we create a search space made of a DAG only having `MLP` and `Identity` layers as candidate operations.

```
from dragon.search_space.bricks_variables import (
    mlp_var, identity_var, operations_var, mlp_const_var, dag_var,
    node_var
)
from dragon.search_space.base_variables import ArrayVar
from dragon.search_operators.base_neighborhoods import
    ArrayInterval

# Candidate operations for the DAG: MLP and Identity layers
candidate_operations = operations_var(
    "Candidate operations", size=10,
    candidates=[mlp_var("MLP"), identity_var("Identity")]
)
dag = dag_var("Dag", candidate_operations)

# Output layer with a Softmax activation function for
    classification
out = node_var(
    "Out",
    operation=mlp_const_var('Operation', 10),
    activation_function=nn.Softmax()
)

# Global search space
```

```
search_space = ArrayVar(
    dag, out, label="Search Space", neighbor=ArrayInterval()
)
```

Listing 4.6: Definition of the classification task search space

Finally, the last step is to implement a training and validation strategy to assess the performance of a configuration from the search space. This can be done by splitting the available dataset into a train and validation set.

```
def train_model(model, data_loader):
    # Model training through gradient descent
    loss_fn = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.05)
    model.train()
    for _ in range(2):
        for X, y in data_loader:
            optimizer.zero_grad()
            y = y.squeeze()
            pred = model(X)
            loss = loss_fn(pred, y)
            loss.backward()
            optimizer.step()
    return model

def validate_model(model, data_loader):
    # Compute the prediction of the trained model on a validation
    set
    loss_fn = nn.CrossEntropyLoss()
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in data_loader:
            y = y.squeeze(1)
            pred = model(X)
            loss = loss_fn(pred, y).item()
            test_loss += loss
            prediction = pred.argmax(axis=1)
            correct += (prediction == y).sum().item()
    accuracy = correct / len(data_loader.dataset)
    return accuracy

def loss_function(args, idx, *kwargs):
    labels = [e.label for e in search_space]
    args = dict(zip(labels, args))
    model = MetaArchi(args, input_shape=(16,))
    model = train_model(model, train_loader)
    accuracy = validate_model(model, val_loader)
    print(f"Idx = {idx}, loss = {1 - accuracy}")
    # Return the loss
    return 1 - accuracy, model
```

```

loss, model = loss_function(search_space.random())
```

Listing 4.7: Model training, validation, and loss computation

The function `loss_function` can be passed to any DRAGON search algorithm as the `evaluation` function. The `search_space` object can be passed as the `search_space` argument.

```
from dragon.search_algorithm.ssea import SteadyStateEA

search_algorithm = SteadyStateEA(search_space, n_iterations=20,
    population_size=5, selection_size=3, evaluation=loss_function,
    save_dir="save/test_ssea/")
search_algorithm.run()
)
```

Listing 4.8: Definition of the DAG-based search space

4.6 Conclusion

In this chapter, we have seen how the search presented in Chapter 2 and the search algorithms from Chapters 2 and 3 are implemented as a Python package called DRAGON. We have described the package's main objects: variables build search spaces, search operators, and search algorithms. We have highlighted what is already available and ready to use, what can be adapted, and what needs to be implemented by the user. Although the package may be difficult to understand for someone new to machine learning, it is quite similar to the *Pytorch* package and can be easily handled by its users. Only the main features of DRAGON are described here. The implementation of each function is described in the documentation [4].

Since the idea behind DRAGON is to provide tools that can be adapted to a wide range of AutoDL problems, it was challenging to design a "no-code" package. However, the tools in the package can be used to build such a framework for specific applications. In Part III of the manuscript, we show how DRAGON can be used to create such frameworks for predicting electricity consumption and wind production, with the creation of EnergyDragon and WindDragon respectively. These frameworks are "no-code" packages that can be used by machine learning novices.

[4] https://dragon-tutorial.readthedocs.io/en/latest/

Part III

Applications to the energy sustainability

Chapter 5

Automated Deep Learning for load forecasting

This chapter presents an application of the DRAGON package to create an Automated Deep Learning framework for load forecasting. Specifically, we tackle the task of forecasting a country's consumption using only external factors, such as weather and calendar variables, without feeding the model with historical data for the target value. This type of model is particularly useful for production planning when there are data stream problems. This forecasting problem is very similar to a regression problem. While regression-based models, in particular Generalized Additive Models (GAMs), are currently effective, the emergence of new explanatory variables and the need to refine the temporality of the signals to be forecasted encourages the exploration of new methodologies, in particular deep learning models. However, Deep Neural Networks (DNNs) struggle with this task due to the lack of data points and the different types of explanatory variables (e.g., integer, float, or categorical). In this chapter, we explain why and how we used Automated Deep Learning (AutoDL) to find performing DNNs for load forecasting. In the end, we created an AutoDL framework called EnergyDragon[1] by using the tools from the DRAGON package presented Chapter 4 and applying them to load forecasting. EnergyDragon automatically selects the features embedded in the DNN training in an innovative way and optimizes the architecture and the hyperparameters of the networks. We demonstrate on the French load signal that EnergyDragon can find original DNNs that outperform state-of-the-art load forecasting methods as well as other AutoDL approaches.

Keisler, J., Claudel, S., Cabriel, G. and Brégère, M. **Automated Deep Learning for load forecasting** *International Conference on Automated Machine Learning (AutoML)* 2024.

[1]The code is available here: https://github.com/JulieKeisler/automl.git.

Contents

5.1 Introduction

Currently, large-scale electricity storage is expensive and relies on inefficient systems. To ensure the safety and smooth operation of the electricity system, it is critical to maintain a strict balance between production and load at all times. Managing this balance relies primarily on the flexibility of programmable power plants which can anticipate electricity demand and adjust their activity accordingly. Load forecasting is essential to program these power plants and to ensure grid stability. Every year, power system operators need forecasting models to provide them with load trends for the coming year and to serve as the basis for short-term forecasts. These models are based on various explanatory variables such as weather (temperature in particular has a strong impact on load) or calendar variables (e.g. load tends to vary between weekdays and weekends). Historical load can be used as a target to train these models for earlier periods, but the one-year forecast horizon makes it unusable as a model input. For this reason, statistical and machine learning methods typically used in time series forecasting are not efficient for this problem. Regression methods, on the other hand, work very well. Over the year, these initial models are then "re-calibrated" with adaptive online learning methods (e.g., online expert aggregation, see Gaillard 2015 or Kalman filter, see Vilmarest 2022) using the lagged data as it becomes available. For example, the re-calibration can be used for day-ahead forecasting to help scheduling production resources for the next day. The re-calibration part is beyond the scope of this chapter, which focuses on the stationary model.

The models used in industry and winning load forecasting competitions (see Farrokhabadi et al. 2022 for a recent one) are regression-based models such as Generalized Additive Models (GAMs) or tree-based models. However, to improve performance and

robustness, and to respond to new industrial challenges such as the integration of new data or the need to forecast at increasingly finer time steps, interest is growing in deep neural networks (DNNs). This is a natural step, as DNNs have proven to be highly effective in fields such as computer vision and natural language processing (NLP). The literature on load forecasting with DNNs mainly approaches it from a time series point of view, using recurrent networks on recently lagged load, which is not applicable in our case. Moreover, DNNs are known to be poorly efficient on tabular regression (Grinsztajn et al., 2022). In our case, the lack of available data (compared to computer vision or NLP datasets, for example) is an additional challenge. The variables used as inputs to the models also have a major impact on performance and may be different from those that work well for the regression models. Nevertheless, we were able to create a DNN with a specific set of explanatory variables that achieves good performance while being slightly below the state of the art. We turned to Automated Deep Learning (AutoDL) to improve on this first model.

In this chapter, we explain how we were able to effectively use AutoDL for load forecasting. We tested several existing methods in the literature, which could not compete with the state-of-the-art, and finally developed our own AutoDL framework: EnergyDragon. It uses the search space of the DRAGON package presented Chapter 4, but includes some innovations such as an original feature selection efficient for load forecasting and a faster search algorithm. Our framework makes it possible to find DNNs that outperform the state of the art in load forecasting by optimizing both their architectures and hyperparameters. We demonstrate its performance on an industrial use case: French load. Finally, we designed EnergyDragon to be understandable and appealing to load forecasting experts who may be new to deep learning. In summary, our contributions are as follows:

- An explanation of our strategy for applying AutoDL to a real-world application, namely load forecasting.
- The AutoDL framework EnergyDragon, an application of DRAGON for load forecasting applications.
- A new feature selection method, embedded in the training of DNNs, that is efficient for load forecasting.
- An application of our results to a concrete use case: the French load forecasting. We show that our approach outperforms the state-of-the-art and other AutoDL techniques.

We begin this chapter by presenting in Section 5.2 why and how we applied AutoDL to load forecasting and position ourselves with the literature. In Section 5.3, we introduce the design of EnergyDragon, an AutoDL framework for load forecasting. Finally,

Section 5.4 details our experimental results obtained on a real-world use case: the forecast of the French load. Section 5.5 concludes the chapter and presents further research opportunities.

5.2 Deep Learning and AutoDL for load forecasting

The load signal can be explained almost entirely by a set of explanatory variables that do not include past data. Therefore performing models tend to be based on regression rather than time series techniques. Multiple linear regressions (MLRs) can be used to calculate the relationships between multiple variables. However, the relationships between load and some exogenous variables are not linear and these models require the specification of functional forms for these variables. The generalized additive models (GAMs) for example, model the nonlinear effects using a spline basis (Pierrot and Goude, 2011). These models, highly accurate for load forecasting, are used in industry and have won several competitions (see for example Nedellec et al. 2014). In this chapter, we are interested in DNNs for load forecasting. Many existing works are based on a setting where past load is immediately available and use time series techniques. For example, Sehovac and Grolinger (2020) uses a sequence-to-sequence recurrent network on historical load with data every five minutes, Rahman et al. (2018) and Mamun et al. (2019) use LSTM (Long-Short Term Memory) models on lagged data and temperature for day-ahead forecasting. Novaes et al. (2021), Zhou et al. (2021) and L'Heureux et al. (2022) tried a transformer-based load forecaster using historical and calendar data for residential load data. Other works, closer to our setting, use DNNs with more explanatory variables or for longer forecast horizons. For example, Farsi et al. (2021) and He (2017) use parallel LSTM/CNN (Convolutional Neural Network) models with different forecast horizons and features, and del Real et al. (2020) forecasts the French load using temperature grids and calendar features as inputs within a CNN. Among all the proposed models, we built a competitive DNN based on CNN and MLP (Multi-Layer Perceptron) layers, called CNN/MLP in the following, whose architecture is closed to Farsi et al. (2021) and He (2017) (we detail this architecture in Appendix C.1.1).

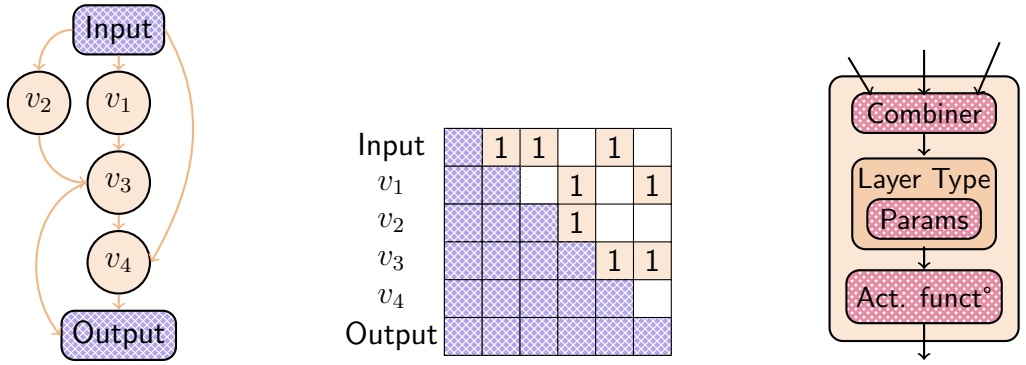
Finding better DNNs for a given task can be done with Automated Deep Learning. AutoDL is a branch of Automated Machine Learning (AutoML) whose goal is to automatically find the best possible DNN for a given problem. AutoDL itself consists of two subproblems, the search for the best architecture, called Neural Architecture Search (NAS), and, for a fixed architecture, the search for the best hyperparameters, called HyperParameters Optimization (HPO). NAS approaches require the definition of a good search space representing all possible solutions. Most search spaces from the literature (Hutter et al., 2019) offer to optimize architectures suitable for computer vision tasks based on CNN layers, pooling layers, and skip connections. Closer to our setting, the AutoPytorch framework has been introduced for tabular (Zimmer et al., 2021) and time series (Deng et al., 2022) data. We tested this framework on our problem (see Sec-

tion 5.4), which could not beat the CNN/MLP model. Then, inspired by Chen et al. (2024) and their work on NAS for multivariate time series forecasting, we used the DARTS (for Differential-Architecture Search, see the original paper Liu et al. 2018d) to relax our original architecture (our search space is given Appendix C.2). Encouraged by the good results of DARTS, we further relaxed our search space using the DRAGON framework proposed Chapter 4, originally introduced for time series forecasting. Compared to the DARTS approach, where the number of layers and the hyperparameters are fixed, the search space defined by DRAGON is more flexible (see Section 5.3.1).

Based on this framework, we created EnergyDragon, which uses the search space of DRAGON, but includes candidate operations specifically designed for load forecasting (see Appendix C.1.2) as well as an innovative feature selection method. The CNN/MLP based architecture depends highly on the input variables. Most AutoDL approaches do not address this issue, which is irrelevant in computer vision or NLP. Surprisingly, neither does Auto-Pytorch, while Grinsztajn et al. (2022) identified the lack of robustness of models to non-informative features as one of the reasons why DNNs perform poorly on tabular data compared to tree-based models. Outside the AutoDL community, feature selection is a widely discussed topic in the literature. Typical approaches include filter methods, wrapper methods, and embedded methods (Li et al., 2017). Filter methods select features based on statistical measures. Wrapper methods train the models with multiple subsets of features and evaluate the features importance based on performance. They are more computationally expensive than filter methods, but can be more efficient. Finally, embedded methods integrate feature selection into the model training process by penalizing the contribution of less important features. In this work, we took inspiration from the DARTS framework and developed our own embedded method, which is described in detail in Section 5.3.2.

5.3 EnergyDragon

In this section, we describe EnergyDragon, our DNNs optimization framework for load forecasting. Section 5.3.1 briefly presents the search space used, which is built using tools from Chapter 4. Next, the following subsections details our contributions to the original framework, adapting it to load forecasting. In Section 5.3.2, we present the objective function. It covers not only network evaluation, but also the feature selection. Due to the specific setting used for the load forecasting task, different from the time series used in Chapter 2, we had to constrained the search space defined Section 5.3.1 using a meta-architecture presented Section 5.3.3. Finally, Section 5.3.4 introduces our search algorithm, an asynchronous evolutionary algorithm.



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

Figure 5.1: DNN encoding as a directed acyclic graph (DAG), as proposed by Chapter 2.

5.3.1 Search Space

The search space used in our framework was presented Chapters 2 and 4 and is defined as $\Omega = (\mathcal{A} \times \{\Lambda(\alpha), \alpha \in \mathcal{A}\})$, where $\mathcal{A}$ is the set of all considered architectures and $\Lambda(\alpha)$ is the set of all considered hyperparameters induced by the architecture α. Each architecture $\alpha \in \mathcal{A}$ is represented by a DAG Γ, where the nodes are the DNN layers and the edges are the connections between them (see Figure 5.1a). The graph adjacency matrix $\mathbf{M} \in \mathbb{R}^{m \times m}$ is used to encode Γ, where m is the number of nodes (see Figure 5.1b), along with a sorted list containing the node hyperparameters $\mathbf{L}$, where $|\mathbf{L}| = m$. In summary, $\mathcal{A} = \{\Gamma = (\mathbf{M}, \mathbf{L})\}$. Each architecture $\alpha \in \mathcal{A}$ induces a hyperparameter search space $\Lambda(\alpha)$. The chosen hyperparameters of all layers from an architecture α are placed in a vector denoted as $\lambda \in \Lambda(\alpha)$. As shown in Figure 5.1c, the layer type: convolution, recurrence, identity, etc., belongs to the architecture search space $\mathcal{A}$, but the layer type parameters: filter size, output shape, etc., the combiner, and the activation function are part of $\Lambda(\alpha)$. The combiner is a function used to combine the multiple inputs of the node. The architecture search space allows multiple input connections, and the incoming vectors can have different shapes. They are combined by the combiner. See Chapters 2 and 4 for more information about this search space.

5.3.2 Objective function

Our objective is to find the DNN $\hat{f} \in \Omega$ having the lowest forecast error on a given load signal. We consider a load dataset $\mathcal{D}$, containing the load signal and the explanatory variables. For any subset $\mathcal{D}_0 = (X_0, Y_0)$, the objective function ℓ is defined as:

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

where the explicit formula for ℓ depends on the task and will be explicitly given Section 5.4. Each DNN $f \in \Omega$ is parameterized by:

- $\alpha \in \Lambda$, its architecture, optimized by the framework.
- $\lambda \in \Lambda(\alpha)$, its hyperparameters, optimized by the framework, where $\Lambda(\alpha)$ is induced by α.
- $\delta \in \Delta(\alpha, \lambda)$, the DNN weights, where $\Delta(\alpha, \lambda)$ is generated by α and λ and optimized by gradient descent when training the model.

In the following, N represents the number of days in the data set (i.e: the number of data samples), H the number of time steps within a day, and F the number of available explanatory variables. The data set $\mathcal{D} = (X, Y)$ consists of $Y = \{\mathbf{y}_t\}_{t=1}^{N} \in \mathbb{R}^{N \times H}$ the target variable and $X = \{\mathbf{x}_t\}_{t=1}^{N} = \{\mathbf{x}_i\}_{i=1}^{F} \in \mathbb{R}^{N \times H \times F}$ the explanatory variables. The optimization aims to find an optimal subset of explanatory variables:

$$\hat{X} = \{\mathbf{x}_j\}_{j \in \mathcal{P}(\{1,\dots,F\})} \subseteq X.$$

To do this, we introduce $\forall j \in [\![1, F]\!] : p_j \in \{0, 1\}$ such that

$$\mathbf{x}_j \in \hat{X} \Leftrightarrow p_j = 1\,.$$

To use gradient descent to find the optimal features, our indicators $p = (p_1, \dots, p_F)$ are relaxed:

$$w = \{\mathrm{sigmoid}(w_j)\}_{j=1}^{F} \in [0, 1]^F \quad \text{with} \quad w_j \in \mathbb{R} \quad \text{and} \quad p_j = \mathbb{1}_{w_j > 0}\,.$$

We partition our time indexes into three groups of successive time steps and split accordingly $\mathcal{D}$ into three datasets: $\mathcal{D}_{\text{train}}$, $\mathcal{D}_{\text{valid}}$, and $\mathcal{D}_{\text{valid}}$. After choosing an architecture α and a set of hyperparameters λ, the DNN $f^{\alpha,\lambda}$ is built and trained on $\mathcal{D}_{\text{train}}$ with respect to a training loss ℓ_{train}. The training of $f^{\alpha,\lambda}$ is divided into two parts. We consider $E = E_w + E_\delta$ as the total number of training epochs. The number of epochs when the feature vector w and the weights δ are jointly optimized is called E_w. Starting at epoch $E_w + 1$, w is transformed to p using the equation $p_j = \mathbb{1}_{w_j > 0}$. Then δ is optimized until the end of the training. Two different losses are used during the training. In the first part, when the current epoch $e \leq E_w$, an L1 penalty is added to ℓ_{train}, like in the LASSO regression (Tibshirani, 1996), to restrain the number of selected features. We define the joint model and features loss $\tilde{\ell}_{\text{train}}$ as:

$$\begin{aligned}\tilde{\ell}_{\text{train}} &: \Omega \times [0, 1]^F \times \mathcal{D} \to \mathbb{R} \\ &f_\delta^{\alpha,\lambda} \times w \times \mathcal{D}_0 \mapsto \ell_{\text{train}}\big(f_\delta^{\alpha,\lambda}(\mathcal{D}_0)\big) + \epsilon \times \sum_{i=1}^{F} |w_i|\,.\end{aligned}$$

When the current epoch $e < E_w$, the training dataset is used to select the best features:

$$\hat{w} \in \underset{w \in [0,1]^F}{\operatorname{argmin}} \left(\min_{\delta \in \Delta(\alpha,\lambda)} \left(\tilde{\ell}_{\text{train}}\left(f_\delta^{\alpha,\lambda}, w, (X_{\text{train}} w, Y_{\text{train}})\right)\right)\right).$$

As $e = E_w$ is reached, $X_{\text{train}}\hat{w}$ is converted to $\hat{X}_{\text{train}}$ and optimize the model weights during the last epochs:

$$\hat{\delta} \in \underset{\delta \in \Delta(\alpha,\lambda)}{\operatorname{argmin}} \left(\ell_{\text{train}}\left(f_\delta^{\alpha,\lambda}, (\hat{X}_{\text{train}}, Y_{\text{train}})\right)\right).$$

The objective function with the DNN parameterized by $\hat{\delta}$ on $\mathcal{D}_{\text{valid}}$ is used to assess the performance of the selected α and λ. The architecture and hyperparameters are optimized using:

$$(\hat{\alpha}, \hat{\lambda}) \in \underset{\alpha \in \mathcal{A}}{\operatorname{argmin}} \left(\underset{\lambda \in \lambda(\alpha)}{\operatorname{argmin}} \left(\ell\left(f_{\hat{\delta}}^{\alpha,\lambda}, (X_{\text{valid}}\hat{w}, Y_{\text{valid}})\right)\right)\right).$$

Finally, the framework output is the objective function with the best architecture, hyperparameters, weights and features on the test dataset:

$$\ell\left(f_{\hat{\delta}}^{\hat{\alpha},\hat{\lambda}}, (X_{test}\hat{w}, Y_{test})\right).$$

Chapter 2 highlighted that the DNNs produced by DRAGON were quite unstable. To fix this, our DNNs are trained with a cyclic learning rate, as suggested by Huang et al. (2022), during the second part of DNN training (when $E_w < e \leq E_w + E_\delta = E$). When the learning rate is low, the neural network reaches a local minimum. We store its weights at that moment, creating an intermediate model. Immediately after that, the learning rate increases again, bringing the model out of the local minimum. At the end of training, the forecasts of the best intermediate models are averaged.

5.3.3 Meta-Architecture

Each DNN $f \in \Omega$ should map an input $X \in \mathbb{R}^{B \times H \times F}$ into a target $Y \in \mathbb{R}^{B \times H}$, where B represents the size of the batch. The generic search space defined Section 5.3.1 should be restrained to architectures that map a two-dimensional input to a one-dimensional output. Therefore, each model $f^{\alpha,\lambda} \in \Omega$ consists in two DAGs Γ_1 and Γ_2. The graph Γ_1 is made of two-dimensional layers operations to treat the matrix X and is parameterized by α_1 and λ_1. A flattened layer follows Γ_1 to transform the two-dimensional latent representation into a one-dimensional one. The graph Γ_2 is then made of one-dimensional layers operations and is parameterized by α_2 and λ_2. We have $\alpha = [\alpha_1, \alpha_2]$ and $\lambda = [\lambda_1, \lambda_2]$. A final output layer maps the output shape of Γ_2 to H. The operations that

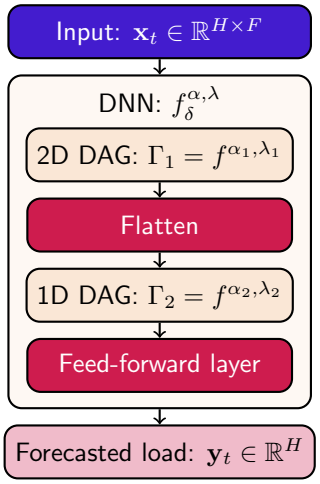


Figure 5.2: Daily meta-model for load datasets.

can be chosen at the nodes of Γ_1 and Γ_2 are given Appendix C.3. A representation of the meta-model is displayed Figure 5.2.

5.3.4 Search Algorithm

In this work, we used an asynchronous (or steady-state) evolutionary algorithm (SSEA) as our search algorithm. However, the framework is implemented so that other search algorithms can be used. In our experiments (see Section 5.4), SSEA is compared with simple random search (RS). Training a DNN is very expensive in terms of time and computational resources. We have access to HPC (High-Performance Computing) resources and exploit them by using the steady-state algorithm introduced Chapter 2. At the beginning of the algorithm, a set of K random DNNs is generated. They will all have small architectures with a small number of layers m. The idea is to start the optimization with simple DNNs to reduce the chance of ending up with overly complex, heavy and unstable DNNs. The weights of the initial features δ can be initialized to uniform random values, zeros, or ones. Then, each initial solution is trained and evaluated on $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{valid}}$ to create our population of size K. For a certain number of iterations T, once a processus is free, two solutions from our population are chosen using a tournament selection. We use the crossover and mutation operators suggested by Chapter 2 to create two offspring that would be 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.

5.4 Experiments

In this section, EnergyDragon performance are evaluated on the French load from March 2019 to March 2020, just before the first lockdown. We have used a rather old year because it is the last year with a stable regime. Since this year, the French load has experienced large perturbations during the COVID lockdowns or the energy crisis. Comparing the performance of steady-state models, which is the subject of this chapter, over periods that are too volatile, without a re-calibration mechanism, is irrelevant. This issue is further discussed Section 5.5.

5.4.1 Dataset

The dataset comes from the website of the French Transmission System Operator[2] (RTE) and contains the French national load data at half-hourly intervals. Therefore, each day contains $H = 48$ time steps. We trained our models from March 2015 to March 2019 and compare the performance from March 2019 to March 2020. We used the Mean Squared Error as ℓ_{train} to optimize the models weights and the Mean Absolute Percentage Error (MAPE) as ℓ. Given a load series $Y = (\mathbf{y}_1 \dots \mathbf{y}_N)$ and the forecasts $\hat{Y} = (\hat{\mathbf{y}}_1, \dots \hat{\mathbf{y}}_N)$, we have:

$$\text{MAPE}(Y, \hat{Y}) = \frac{1}{N} \sum_{i=1}^{N} \left| \frac{\mathbf{y}_i - \hat{\mathbf{y}}_i}{\mathbf{y}_i} \right|.$$

For this dataset thirty-four explanatory variables can be used. The weather data contains the national temperature along with exponential smoothing variants of parameters going from 0.7 to 0.998, wind and cloud cover. Calendar features include the day of the week, the month, the year, if the day correspond to a public holidays or a surrounding day.

5.4.2 Baseline

We compare our results to models at the state-of-the-art in load forecasting: a Generalized Additive Model (GAM) used in the industry, the CNN/MLP model and to two AutoML/AutoDL approaches: AutoPytorch (Zimmer et al., 2021) and a version of DARTS (Liu et al., 2018d) applied on the handcrafted DNN. AutoPytorch includes the hyperparameters tuning, model selection and ensembling of simple regression models such as Random Forest, Support Vector Machine (SVM) or Catboost for example, therefore they are not directly included in the baseline.

Generalized Additive Model The GAMs are state of the art for load forecasting (see among others Pierrot and Goude 2011 or Wood 2017). The output Y is explained

[2] https://www.rte-france.com/eco2mix

as the sum of smooth functions of the explanatory variables: $Y = g_1(X_1) + g_2(X_2) + g_3(X_3, X_4) + \dots$ where the g_i are linear or a special type of piecewise polynomial functions called splines. The GAM developed for this use case is instantaneous, meaning that a model is fitted for each of the 48 time steps of the day. For our experiments, we chose a GAM that is used in the industry and therefore cannot reveal its explicit formula. The GAM takes about twenty explanatory variables as input.

Deep Neural Networks We included our CNN/MLP inspired by Farsi et al. (2021) and He (2017) in our baseline. Unlike the GAM, a single model is used to predict the 48 time steps of the day. The input variables are divided into two groups of about ten features. One group is processed by parallel one-dimensional convolutions and the other by feed-forward layers. The branches are then concatenated and processed by more feed-forward layers. The detailed architecture can be found Appendix C.1.1. The model was trained with the Adam optimizer (Kingma and Ba, 2017) during 500 epochs.

AutoPytorch We used the tabular regression API of AutoPytorch[3]. This framework combines an AutoML pipeline for traditional regression models (e.g., RandomForest, CatBoost or LightGBM) with the tuning of DNNs. The search-space used for the AutoDL part is made of MLPs, residual connections and Normalization Layers. The framework does not allow us to map two-dimensional inputs to one-dimensional targets, so each moment of the day was forecasted independently. The hour of the day and the instant were added as explanatory variables. For a fair comparison, the same global optimization budget was set for both AutoPytorch and EnergyDragon: 24 hours, and the same budget by model: 15 minutes. Two versions are used in the baseline: with the traditional regression baseline and with only the AutoDL part (fairer with EnergyDragon).

DARTS We optimized the CNN/MLP architecture using the DARTS (Liu et al., 2018d) algorithm. The search space is described in detail in Appendix C.2. In short, parts of the original architecture are replaced with DARTS cells. Each cell is a DAG where the links represent candidate operations. During optimization, multiple operations are considered for each link and are associated with a probability of being selected. These probabilities are optimized by gradient descent. At the end of the optimization, the operation with the highest probability is chosen in the final architecture. The model weights and the operation probabilities are optimized alternatively, with 500 epochs for the weights and 200 epochs for the probabilities, using the Adam optimizer Kingma and Ba (2017).

EnergyDragon For EnergyDragon (hereafter called ED),the global time budget is fixed to 24 hours. The features are optimized during 500 epochs and the weights during

[3] https://github.com/automl/Auto-PyTorch/tree/master

200 epochs. For the steady-state evolutionary algorithms, the initial population size is set to $K = 100$. A DNN cannot be trained for more than 15 minutes. The baseline compared five versions of ED. One with a random search algorithm, called ED RS, the other versions use the steady-state evolutionary algorithm. ED SSEA is implemented with the mutation operators of DRAGON but without the crossover, and ED SSEA Crossover uses the crossover. Finally, ED SSEA CNN/MLP and ED SSEA Crossover CNN/MLP include the MLP/CNN model in the initial population. This means that 99 models are randomly initialized and the remaining one is the CNN/MLP model.

5.4.3 Results

Model	MAPE	RMSE (in MW)
GAM	1.398%	929.8
AutoPytorch	17.999%	10641.7
AutoPytorch with the traditional baseline	2.022%	1243.2
CNN/MLP (handcrafted DNN)	1.721%	1164.6
DARTS	1.600%	1085.6
ED RS	1.374%	902.3
ED SSEA	1.258%	851.4
ED SSEA Crossover	1.190%	837.8
ED SSEA Crossover CNN/MLP	1.182%	816.3
ED SSEA CNN/MLP	**1.131%**	**803.4**

Table 5.1: MAPE and RMSE of the different models from our baseline. The reference model is the GAM and the best model is highlighted in bold.

We evaluated each algorithm from the baseline on the French load signal. Each version from ED was run using 20 GPUs V100, and AutoPytorch using 2 Quadro RTX 6000 which are faster than the V100. We used in total approximatively 336 GPU-hours. Each algorithm was run with a global seed of 0 to ensure reproducibility. The results can be found in Table 5.1. In addition to the MAPE function, the Root Mean Squared Error is (RMSE) is also reported. For all proposed versions, the results of the EnergyDragon algorithms beat all other models from the baseline. The AutoPytorch framework had the worst results, even with the traditional models. The lack of feature selection may explain these results. Our CNN/MLP handcrafted model was slightly improved by the DARTS framework, but both versions cannot compete with the reference model (GAM). Among the ED results, the random search got the worst results, which demonstrates the performance of our search algorithm (see Appendix C.4.3 for convergence plots). Although the CNN/MLP model does not outperform the GAM, it is still useful to use it as an input for ED. In fact, both versions with and without crossover with the CNN/MLP in

the initial population outperformed the versions without. Finally, the crossover helped to improve the performance of ED without the CNN/MLP as input, but the best version of ED was with the CNN/MLP and without the crossover. The initial population of this last algorithm already contains a good candidate (the CNN/MLP model), and therefore does not need as much exploration (with the crossover) as the version without the CNN/MLP. The models found by EnergyDragon for each setup can be found in Appendix C.4.1. The best version of EnergyDragon: ED SSEA CNN/MLP improves the predictions of GAM by 19%. Figure 5.3 shows the forecast of GAM and ED SSEA CNN/MLP for the last week of November. Forecasts from other algorithms can be found in Appendix C.4.2. The forecast signals have similar shapes for GAM and ED SSEA CNN/MLP, but GAM has a larger bias and overpredicts the load. More details on the experimental results can be found in Appendix C.4, as well as another case study on the Norwegian load Appendix C.5.

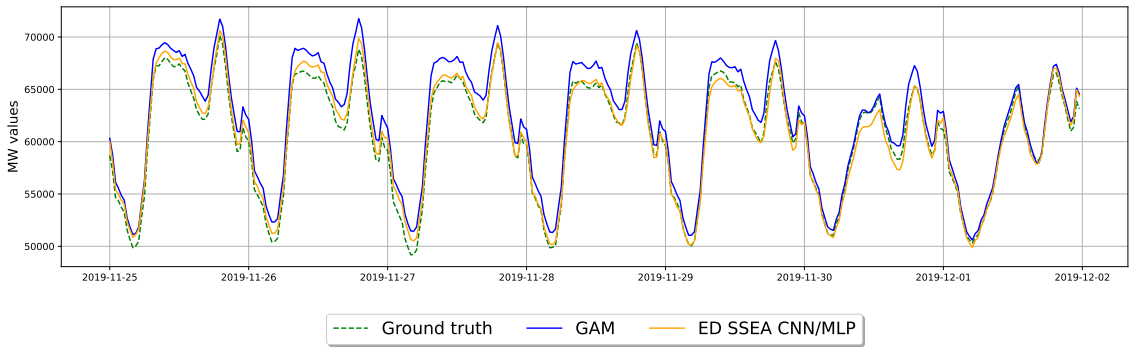


Figure 5.3: Load power forecasting for the last week of November 2019. The ground truth is displayed in dotted line, the GAM forecast is drawn with a blue line whereas the forecast from the best version of EnergyDragon (ED SSEA CNN/MLP) is drawn in yellow.

5.5 Conclusion

This chapter explains how we applied AutoDL to a real-world application: load forecasting. The existing works in the AutoDL community were not sufficient to be used directly in our case, and we had to develop our own AutoDL framework, EnergyDragon. This framework is able to automatically select input features and optimize DNN architectures and hyperparameters to generate performing models. We demonstrate on the French load signal that EnergyDragon is able to outperform a state-of-the-art model in load forecasting: a generalized additive model used in the industry as well as an AutoML framework designed for tabular regression: AutoPytorch. Future work should focus on automatically re-calibrating the models found by EnergyDragon so that they can be used for short-term forecasting in rather erratic periods. In addition, the industry requires the

interpretability of the forecasting models. DNNs are known to be black boxes, and to be accepted as an industrial solution, we will have to work on finding ways to interpret their forecasts.

Chapter 6

Automated Spatio-Temporal Weather Modeling for Load Forecasting

In this chapter, we further optimize the pipeline started with EnergyDragon for automated load forecasting introduced Chapter 5. The load signal is highly dependent on meteorological variables (temperature, wind, sunshine). These dependencies are complex and difficult to model. On the one hand, spatial variations do not have a uniform impact because population, industry, and wind and solar farms are not evenly distributed across the territory. On the other hand, temporal variations can have delayed effects on load (due to the thermal inertia of buildings). With access to observations from different weather stations and simulated data from meteorological models, we believe that both phenomena can be modeled together. In today's state-of-the-art load forecasting models, including EnergyDragon introduced Chapter 5, the spatio-temporal modeling of the weather is fixed. In this chapter, we aim to take advantage of the automated representation and spatio-temporal feature extraction capabilities of deep neural networks to improve spatio-temporal weather modeling for load forecasting. We compare our deep learning-based methodology with the state-of-the-art on French national load. This methodology could also be fully adapted to forecasting renewable energy production.

Keisler, J. and Brégère M. **Automated Spatio-Temporal Weather Modeling for Load Forecasting.** *International Ruhr Energy Conference (INREC), Best paper award*, 2024.

Contents

6.1 Introduction

The cost of large-scale electricity storage remains high, and the current systems in use remain inefficient. Furthermore, the secure and smooth operation of the power grid depends on maintaining a constant and precise balance between electricity production and demand. The aforementioned equilibrium is achieved via the adaptability of programmable power plants, which modify their production in accordance with load forecasts. It is therefore essential to have accurate forecasts of both electricity demand and the output of renewable energy sources in order to schedule power plants and maintain grid stability. The two signals depend on meteorological variables, specifically temperature, wind speed, and solar radiation, which vary in both space and time. As consumer demand and renewable energy generation facilities are not evenly distributed across a given area, variations in meteorological conditions at a particular location will affect these signals. In addition, temporal weather variations can have a delayed effect, particularly with regard to the load, due to the thermal inertia of buildings and the reactivity of consumers. It can be assumed that the manner in which temporal and spatial variations in weather patterns are modelled has a significant impact on the efficacy of the forecasting models.

This chapter concentrates on short-term load forecasting with a forecast time horizon of one day. Such forecasts enable power system operators to make adjustments to production and spot market prices. This signal is challenging to forecast due to its dependence on a multitude of variables, including meteorological factors (temperature, wind, etc.) and calendar-related elements (holidays, weekdays, etc.). Consequently, the models employed in the industry and those that have been successful in load forecasting competitions (see Farrokhabadi et al. 2022, for a recent example) are regression-based models, such as Generalized Additive Models (GAMs) or tree-based models. In general,

lagged load is not employed. While this variable offers valuable insight, it can also limit the model's interpretability by reducing the importance of other variables. Furthermore, the model would become unusable in case of data stream issues. To address this challenge while leveraging the insights offered by lagged load, a promising approach is to construct a static model only based on the explanatory variables and then recalibrate this model by adjusting certain parameters a posteriori using lagged load. For instance, Ba et al. (2012b) proposes adaptive learning algorithms that combine additive models with a recursive least squares filter, while Vilmarest (2022) employs a Kalman filter to perform an online adaptation of the weights of their models.

Despite their impressive performance in various fields, such as computer vision and natural language processing, deep neural networks (DNNs) are still not widely used in the load forecasting community. However, recent work presents promising results for load forecasting using DNNs. In particular, Chapter 5 proposed EnergyDragon, a deep neural network optimization framework designed for load forecasting. EnergyDragon automatically finds high-performance neural networks for the static part of load forecasting models and is able to outperform state-of-the-art regression models. Since neural networks have demonstrated their ability to extract relevant features from data in a variety of formats, we thought it would be interesting to try them on raw spatio-temporal weather data to see if they could automatically find more relevant spatio-temporal representations than the fixed ones used in the state of the art. In summary, this chapter contributions are as follows:

- A DNN-based spatio-temporal weather modeling for load forecasting, which improves on the static modeling currently in use while remaining interpretable.
- The integration of this weather modeling approach into the framework EnergyDragon.
- An application of our results to a concrete use case: the day-ahead French load forecasting over a turbulent period: sobriety during the year 2023.

We start this chapter by presenting in Section 6.2 the state of the art in short-term load forecasting: regression-based models, EnergyDragon and recalibration methods. In Section 6.3, we show how to learn the actual spatio-temporal modeling approach with DNN. Section 6.4 introduce how to incorporate the spatio-temporal weather modeling into EnergyDragon. Finally, Section 6.5 details our experimental results obtained on a real-world use case: the French national load forecasting. Section 6.6 concludes the chapter and presents further research opportunities.

6.2 Related Work

The load signal can be explained almost entirely by a set of explanatory variables that do not include the past target data. Consequently, the majority of performing models

are based on regression rather than time series techniques. Multiple Linear Regressions (MLRs) can be used to calculate the relationships between multiple variables. However, the relationships between load and some exogenous variables are not linear, and thus, these models require the specification of functional forms for these variables. For instance, Generalized Additive Models (GAMs) employ a spline basis to model the non-linear effects, as detailed in (Pierrot and Goude, 2011). These models, which are highly accurate for load forecasting, are used in industry and have been the winners of several competitions (see, for example, Nedellec et al. 2014).

DNNs have dominated the fields of computer vision and natural language processing for some years now. They offer the ability to process data in a variety of formats - e.g., text, images, graphs - make them particularly interesting for load forecasting, which depends on a large number of explanatory variables that may come from data sets in a variety of formats. Recently, they have also revolutionized the field of weather forecasting, proving more effective than physic-based forecasting models (see for example Pathak et al. 2022 and Lam et al. 2023). While the initial work was based on gridded reanalysis data, McNally et al. (2024) have shown that DNNs could also be effective in extracting spatio-temporal features directly from raw weather data. In the field of load forecasting, they are currently less widely used than multilinear regression models. However, Chapter 5 we demonstrated that by optimizing the structure and hyperparameters of DNNs, it is possible to develop models that surpass the current state of the art. In this chapter, we optimized DNNs using the DRAGON package presented Chapter 4. The models are represented using Directed Acyclic Graphs (DAGs). The search space is defined as $\Omega = (\mathcal{A} \times \{\Lambda(\alpha), \alpha \in \mathcal{A}\})$, where $\mathcal{A}$ is the set of all considered architectures and $\Lambda(\alpha)$ is the set of all considered hyperparameters induced by the architecture α. Each architecture $\alpha \in \mathcal{A}$ is represented by a DAG Γ, where the nodes are the DNN layers and the edges are the connections between them. See Chapter 5 for more information about this search space.

Finally, this chapter deals with short-term forecasting of electricity consumption. The COVID crisis and recent european energy crises have highlighted the importance of models that can rapidly adapt to new contexts. This is why research in the field has focused on different techniques for online model adaptation. These include the Kalman filter adaptation of Generalized Additive Models (GAMs), which won the post-covid electricity load forecasting competition (see Farrokhabadi et al. 2022 and De Vilmarest and Goude 2022). The adaptation is done by multiplying the GAM effects vector by a linear correction. Let's have $x_t \in \mathbb{R}^D$ our features vector, with $D \in \mathbb{N}^\star$, $y_t \in \mathbb{R}$ the target and $\hat{y}_t \in \mathbb{R}$ the forecasted target. The static GAM model can be defined as:

$$\hat{y}_t = \sum_{i=1}^{D} f_i(x_t).$$

Let's have $f(x_t) = [f_i(x_t)]_{i=1}^D$ the GAM effects vector, the adaptation is done by fitting

a vector $\theta_t \in \mathbb{R}^D$ called state, such that:

$$\hat{y}_t = \sum_{i=1}^{D} \theta_{i,t} f_i(x_t) + \epsilon_t = \theta_t^T f(x_t) + \epsilon_t \tag{6.1}$$

$$\theta_{t+1} = \theta_t + \eta_t, \tag{6.2}$$

where $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$ and $\eta_t \sim \mathcal{N}(0, Q)$, with σ^2 and Q are time-invariant and assumed to be known. The algorithm achieves the estimation of the state θ_t by computing its state posterior distribution as a Gaussian distribution: $\theta_t|(x_s, y_s)_{s<t} \sim \mathcal{N}(\hat{\theta}_t, P_t)$. The algorithm relies on the choice of σ and Q. Vilmarest (2022) suggests an iterative grid search. In this work, we propose to apply a state vector θ_t on the last layer of a DNN and to optimize σ and Q directly through the EnergyDragon framework, as any other hyperparameter.

6.3 Weather Modeling with Deep Neural Networks for load forecasting

In this work, we aim to forecast at each time step $t \in [1, \dots, N]$, with $N \in \mathbb{N}^\star$, a daily load variable $y_t \in \mathbb{R}^H$, using a features vector $x_t = (w_t, c_t) \in (\mathbb{R}^{H \times V \times I} \times \mathbb{R}^{H \times F})$, where N represents the number of days in the data set and H the number of time steps within a day. The features vector x_t is made of two elements: w_t gathering the spatio-temporal weather data and c_t containing the other $F \in \mathbb{N}$ explanatory variables such as calendar data (e.g., months, years holidays). The vector $w_t \in \mathbb{R}^{H \times V \times I}$ contains the forecasts at time t from different weather stations, or from a forecasted weather grid, produced by, for example, a NWP model. The dimension $I \in \mathbb{N}^\star$ corresponds to the number of spatial points (i.e., the number of weather stations in the first case and the number of grid points in the second) and $V \in \mathbb{N}^\star$ to the number of weather variables present in w_t (e.g., temperature, wind speed, solar radiation).

6.3.1 Spatio-Temporal weather modeling

In order to be integrated into load forecasting models, spatio-temporal weather is deterministically transformed into "electrical" weather. Several functions are applied in order to reduce the information and extract what will be most useful for load forecasting. These functions have been defined with industry expertise, but are not adapted to a particular dataset or period. An example of such functions for the french load signal are given by the French Transmission System Operator called RTE[1].

[1] https://www.services-rte.com/files/live/sites/services-rte/files/documentsLibrary/2022-04-01_REGLES_MA-RE_SECTION_2_F_3590_en

Ponderation The first step is to switch back from the multi-variate, spatial signal to an aggregated univariate signal. The I spatial locations are not necessarily evenly distributed throughout the considered region and do not contribute equally to the electrical weather. For instance locations in densely-populated parts have more weights than others located in isolated areas. Let's denote $w_h^{v,i} \in \mathbb{R}$ the forecast of the weather variable v (e.g. temperature) at time step h of the location i, and $a^i \in [0,1]$ the weight of the location i. The weights are shared accross the variables v. The aggregated signal at time h can then be written as:

$$w_h^v = \sum_{i=1}^{I} a^i w_h^{v,i}. \tag{6.3}$$

This behavior can easily be reproduced with a Multi-Layer Perceptron (MLP):

$$w_h^v = A\mathbf{w}_h^v + b, \text{ with } \mathbf{w}_h^v = [w_h^{v,i}]_{i=1}^I,\ A = [a^i]_{i=1}^I \text{ and } b = 0 \tag{6.4}$$

However, a Deep Neural Networks requires the scaling of the input data. Each v variable is scaled independently, so that variables with large amplitudes (e.g. temperature) do not override the others. We scale each location i independently and denote $\tilde{w}_h^{v,i} = (w_h^{v,i} - \min^{v,i})/(\max^{v,i} - \min^{v,i})$ the min-max scaled version of $w_h^{v,i}$, with $\min^{v,i} = \min_{h\in[1,\dots,H]} w_h^{v,i} \in \mathbb{R}$ and $\max^{v,i} = \max_{h\in[1,\dots,H]} w_h^{v,i} \in \mathbb{R}$. If we consider the aggregated target to also be scaled, with $\min^v = \min_{h\in[1,\dots,H]} w_h^v \in \mathbb{R}$ and $\max^v = \max_{h\in[1,\dots,H]} w_h^v \in \mathbb{R}$, we have:

$$\begin{aligned}
\tilde{w}_h^v = \frac{w_h^v - \min^v}{\max^v - \min^v} &= \big(\sum_{i=1}^{I} a^i w_h^{v,i} - \min^v\big)/(\max^v - \min^v) \\
&= \sum_{i=1}^{I} \tilde{w}_h^{v,i}(\max^{v,i} - \min^{v,i}) + \min^{v,i} - \min^v\big)/(\max^v - \min^v) \\
&= \sum_{i=1}^{I} a^i \frac{\max^{v,i} - \min^{v,i}}{\max^v - \min^v} \tilde{w}_h^{v,i} + \sum_{i=1}^{I} \frac{\min^{v,i} - \min^v}{\max^v - \min^v} \\
&= A_v \tilde{\mathbf{w}}_h^v + b_v
\end{aligned}$$

,

with:

- $\tilde{\mathbf{w}}_h^v = [\tilde{w}_h^{v,i}]_{i=1}^I$,
- $A_v = [a^i(\max^{v,i} - \min^{v,i})/(\max^v - \min^v)]_{i=1}^I$,
- $b_v = \sum_{i=1}^I (\min^{v,i} - \min^v)/(\max^v - \min^v)$.

Therefore, we need one MLP to aggregate each of the V weather variables (e.g., temperature or wind speed), resulting in V MLP layers.

6.3.2 Temperature smoothing

Load does not respond instantaneously to changes in the weather. In particular, temperature effects are more gradual due to the thermal inertia of buildings. This is why the concept of smoothed temperature is useful for understanding the factors that influence electricity consumption. Exponential smoothing is typically employed in this context. We denote, for a day t, $\mathcal{T}_t = [\mathcal{T}_{t,1}, \ldots, \mathcal{T}_{t,H}] \in \mathbb{R}^H$ the aggregated temperature and $\overline{\mathcal{T}}_t = [\overline{\mathcal{T}}_{t,1}, \ldots, \overline{\mathcal{T}}_{t,H}] \in \mathbb{R}^H$ the smoothed version. We define:

$$\overline{\mathcal{T}}_{t,1} = (1-\nu)\mathcal{T}_{t,1} + \nu\overline{\mathcal{T}}_{t-1,H}, \text{ and, } \forall i \in [2,H] : \overline{\mathcal{T}}_{t,i} = (1-\nu)\mathcal{T}_{t,i} + \nu\overline{\mathcal{T}}_{t,i-1}, \quad (6.5)$$

where $\nu \in [0,1]$ is the smooth coefficient, which can be optimized.

Recurrent Neural Networks Smoothed temperature requires at each time step t to pass $\overline{\mathcal{T}}_{t,H} \in \mathbb{R}$ to the next time step $t+1$. Such information passing can be reproduced by Recurrent Neural Networks (RNNs), which are designed with a memory vector. The equations of a recurrent neural network with input $\mathcal{T}_t \in \mathbb{R}^H$ and output $\overline{\mathcal{T}}_t \in \mathbb{R}^H$ can be written as:

$$\overline{\mathcal{T}}_t = \phi(\mathcal{T}_t W_1^T + b_1 + \overline{\mathcal{T}}_{t-1} W_2^T + b_2)\,,$$

where ϕ is an activation function (typically non-linear), and $W_1 \in \mathbb{R}^{H\times H}$, $W_2 \in \mathbb{R}^{H\times H}$, $b_1 \in \mathbb{R}^H$ and $b_2 \in \mathbb{R}^H$ are some parameters which can be learned through gradient descent. For writing simplicity, we now index our temperature by $t^\star$, such that, if $t^\star = \{t,i\}$, we have, if $i < H$:

$$\begin{cases} t^\star = \{t, i+1\} \\ \text{else, } t^\star + 1 = \{t+1, 1\}. \end{cases}$$

Let $\tau > 0$. To compute $\overline{\mathcal{T}}_{t^\star}$ based on the smoothed temperature at instant $t^\star - \tau$, $\overline{\mathcal{T}}_{t^\star-\tau}$, and the sequence of temperatures $\mathcal{T}_{t^\star-\tau+1}, \ldots, \mathcal{T}_{t^\star}$, we have:

$$\begin{aligned} \overline{\mathcal{T}}_{t^\star-\tau+1} &= \overline{\mathcal{T}}_{t^\star-\tau} \\ \overline{\mathcal{T}}_{t^\star-\tau+1} &= (1-\nu)\mathcal{T}_{t^\star-\tau+1} + \nu\overline{\mathcal{T}}_{t^\star-\tau} \\ \vdots \quad &= \quad \vdots \\ \overline{\mathcal{T}}_{t^\star} &= \sum_{s=0}^{\tau-1} \nu^s (1-\nu)\mathcal{T}_{t^\star-s} + \nu^\tau \overline{\mathcal{T}}_{t^\star-\tau}\,. \end{aligned} \quad (6.6)$$

Based on Equation (6.6), by setting:

$$W_1 = \begin{bmatrix} (1-\nu) & 0 & 0 & \dots & 0 \\ \nu(1-\nu) & (1-\nu) & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \nu^{H-1}(1-\nu) & \nu^{H-2}(1-\nu) & \nu^{H-3}(1-\nu) & \dots & (1-\nu) \end{bmatrix},$$

$$W_2 = \begin{bmatrix} 0 & \dots & 0 & \nu \\ 0 & \dots & 0 & \nu^2 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & \vdots & 0 & \nu^H \end{bmatrix},$$

$$\text{and } b_1 = b_2 = 0,$$

It is possible to induce a recurrent neural network (RNN) to learn the behaviour defined by the exponential smoothing model, as set out in Equation (6.5). Nevertheless, this configuration results in the network optimizing a total of $\mathcal{O}(H^2)$ parameters. In the context of seeking novel approaches to temperature smoothing, the utilization of a recurrent neural network (RNN) is a logical choice. On the other hand, if the objective is to restrict the network to exponential smoothing, with optimization limited to the ν smoothing coefficient, the necessity for optimizing a vast number of parameters renders the process complex and may ultimately prove inefficient. For this reason, we propose a new DNN layer that enables the efficient computation of one or more exponential smoothings over several batches, with only the smoothing coefficients as parameters to optimize.

Exponential Smoothing Layer Let's consider a batched input of size $B \in \mathbb{N}^\star$, containing the temperature from the days $t-B$ to t: $\mathcal{T}_{t-B:t} = [\mathcal{T}_{t-B}, \dots, \mathcal{T}_t] \in \mathbb{R}^{B \times H}$. The exponential smoothing layer first reshape this data into a size BH, to treat all sequences at once. We then use Equation (6.6), with $\tau = H \times B - 1$ to compute $\overline{\mathcal{T}}_{t-B:t}$:

$$\begin{bmatrix} \overline{\mathcal{T}}_{t^\star-\tau} \\ \overline{\mathcal{T}}_{t^\star-\tau+1} \\ \vdots \\ \overline{\mathcal{T}}_{t} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \nu & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \nu^\tau & \nu^{\tau-1} & \nu^{\tau-2} & \dots & 1 \end{bmatrix} \begin{bmatrix} \overline{\mathcal{T}}_{t^\star-\tau} \\ (1-\nu)\mathcal{T}_{t^\star-\tau+1} \\ \vdots \\ (1-\nu)\mathcal{T}_{t^\star} \end{bmatrix},$$

$$\overline{\mathcal{T}}_{t^\star-\tau:t^\star} = M \times \left[\overline{\mathcal{T}}_{t^\star-\tau} \,|\, (1-\nu)\mathcal{T}_{t^\star-\tau+1:t^\star}\right], \text{with}, \forall i \geq j : M_{i,j} = \nu^{i-j}\,. \tag{6.7}$$

Finally, we reshape $\overline{\mathcal{T}}_{t^\star-\tau:t^\star} \in \mathbb{R}^{HB}$ back to the original shape $\overline{\mathcal{T}}_{t-B:t} \in \mathbb{R}^{H \times B}$. The matrix M is constructed during the forward pass as its shape and formula depend on the batch size B. Given that the coefficient ν belongs to $[0, 1]$, it is encoded through a

sigmoid: $\nu = \text{Sigmoid}(\overline{\nu}) = 1/\big(1+\exp(-\overline{\nu})\big) \in [0,1]$ for $\overline{\nu} \in \mathbb{R}$, where $\overline{\nu}$ would be the weight optimized through back-propagation.

In our search space we let the optimization framework choose between the Exponential Smoothing Layer, a RNN layer, a Long-Short Term Memory (LSTM) layer and a Gated Recurrent Unit (GRU) layer, to perform the smoothing operation (see Section 6.4.2).

6.3.3 Online learning

The last layer of the search space from Chapter 5 is a linear layer, transforming an input $h_t \in \mathbb{R}^{H \times D}$ into an output $y_t \in \mathbb{R}^H$, where y_t is the load consumption for the day t, H the number of instants during the day and $D \in \mathbb{N}^\star$ is the hidden dimension within the network before the last layer. Let's name $A_D \in \mathbb{R}^D$ and $b_D \in \mathbb{R}$ respectively the weights and bias matrices of this last MLP layer, we have:

$$y_t = A_D h_t + b_D = \sum_{i=1}^{D} a_D^i h_t^i + b_D. \tag{6.8}$$

To adapt our DNN, we use a daily Kalman state vector $\theta_t \in \mathbb{R}^D$ to adapt the coefficients of Equation (6.8):

$$\tilde{y}_t = \theta_t^T (A_D h_t + b_D) = \sum_{i=1}^{D} \theta_t^i (a_D^i h_t^i + b_D) + \epsilon_t \tag{6.9}$$

$$\theta_{t+1} = \theta_t + \eta_t, \tag{6.10}$$

where $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$ and $\eta_t \sim \mathcal{N}(0, Q)$. Vilmarest (2022) suggests to use iterative grid search for $\sigma \in \mathbb{R}$ the diagonal coefficients of $Q \in \mathbb{R}^{D \times D}$. This search can be quite expensive, with a complexity of $\mathcal{O}(LD^2)$, where L is number of values that the coefficients of Q may take. We experimented empirically that the number of coefficients of the last MLP layer is usually larger than the number of coefficient of the GAMs. The iterative grid search was not usuable in this case, therefore we included the optimization of σ and the coefficients of Q within EnergyDragon search space (see Section 6.4.2).

6.4 Automated weather modeling

This Section presents the integration of the various elements presented in Section 6.3, namely the weather modeling and Kalman adaptation modules, into EnergyDragon with the objective of optimizing them.

6.4.1 Objective function

The objective is to identify the optimal DNN $\hat{f} \in \Omega$ with the lowest forecast error on a given load signal with a short forecast horizon (e.g., 24 hours). The load dataset, denoted by $\mathscr{D}$, contains the spatio-temporal W data, the explanatory variables C and the target (the load signal) Y. For any subset $\mathscr{D}_0 = \big((W_0, C_0), Y_0\big)$, the objective function ℓ is defined as:

$$\begin{aligned} \ell \colon \Omega \times \mathscr{D} &\to \mathbb{R} \\ f \times \mathscr{D}_0 &\mapsto \ell\big(f(\mathscr{D}_0)\big) = \ell\big(Y_0, f(W_0, C_0)\big). \end{aligned}$$

Each DNN $f \in \Omega$ is parameterized by:

- $\alpha \in \mathscr{A}$, its architecture, optimized by the framework.
- $\lambda \in \Lambda(\alpha)$, its hyperparameters, optimized by the framework, where $\Lambda(\alpha)$ is induced by α. The hyperparameters include Q and σ from the Kalman adaptation. It should be noted that the shape of Q depends on the architecture and hyperparameters of the networks.
- $\delta \in \Delta(\alpha, \lambda)$, the DNN weights, where $\Delta(\alpha, \lambda)$ is generated by α and λ and optimized by gradient descent when training the model.

The optimization process is done in several steps. First, the optimal DNN weights $\hat{\delta} \in \Delta(\alpha, \lambda)$ for a given architecture $\alpha \in \mathscr{A}$ and set of hyperparameters $\lambda \in \Lambda(\alpha)$ are found using gradient descent with respect to the training loss ℓ_{train} over the training set $\mathscr{D}_{\text{train}} = \big((W_{\text{train}}, C_{\text{train}}), Y_{\text{train}}\big) = (X_{\text{train}}, Y_{\text{train}})$:

$$\hat{\delta} \in \underset{\delta \in \Delta(\alpha,\lambda)}{\operatorname{argmin}} \Big(\ell_{\text{train}}\big(f_{\delta}^{\alpha,\lambda}(X_{\text{train}}, Y_{\text{train}})\big) \Big).$$

Once the DNN is trained, the performance of the selected α and λ are evaluated on $\mathscr{D}_{\text{valid}}$ using ℓ. As Q and σ are part of λ, the evaluation is done using the Kalman recalibration of the model. First, the state vector $\theta \in \mathbb{R}^{H \times D}$ is estimated on the last MLP layer of the trained DNN $f_{\hat{\delta}}^{\alpha,\lambda}$ using Equations (6.9) and (6.10). Let's have $\Theta_\lambda\big(f_{\hat{\delta}}^{\alpha,\lambda}(X_{\text{valid}})\big)$ the recalibration of $f_{\hat{\delta}}^{\alpha,\lambda}(X_{\text{valid}})$ as defined Equation (6.9). The architecture α and hyperparameters λ are optimized as:

$$(\hat{\alpha}, \hat{\lambda}) \in \underset{(\alpha,\lambda) \in (\mathscr{A} \times \Lambda(\alpha))}{\operatorname{argmin}} \left(\ell\Big(\Theta_\lambda\big(f_{\hat{\delta}}^{\alpha,\lambda}(X_{\text{valid}})\big), Y_{\text{valid}}\Big) \right).$$

The framework output will be the objective function ℓ computed using the DNN with the best architecture, hyperparameters, weights and calibration using Kalman on the test dataset:

$$\ell\Big(\Theta_{\hat{\lambda}}\big(f_{\hat{\delta}}^{\hat{\alpha},\hat{\lambda}}(X_{\text{test}})\big), Y_{\text{test}}\Big)\,.$$

In the following section (6.4.2), we explicit our search space, defined by $\mathcal{A}$ and $\Lambda(\alpha)$.

6.4.2 Search space

The search space used in this work extends the one from Chapter 5 by adding a weather modeling and a Kalman module, and is depicted Figure 6.1. Each DNN $f \in \Omega$ maps two batched inputs: $w_b \in \mathbb{R}^{B\times H\times V\times I}$ containing the spatio-temporal weather and $c_b \in \mathbb{R}^{B\times H\times F}$ containing the other explanatory variables into a target $y_b \in \mathbb{R}^{B\times H}$, where $B \in \mathbb{N}^\star$ represents the size of the batch.

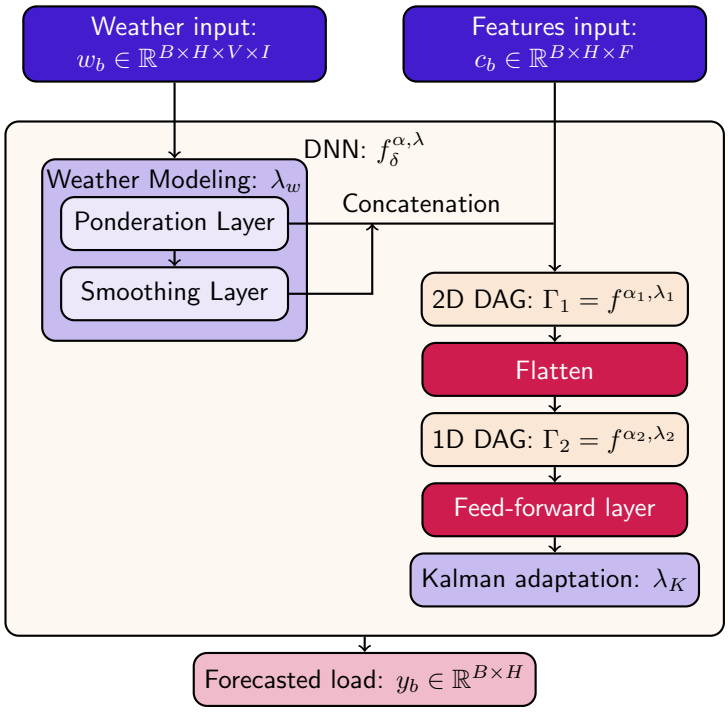


Figure 6.1: Daily meta-model for load datasets from Chapter 5, with the integration of the weather modeling and the Kalman adaptation modules.

Weather Modeling The weather input designated as w_b, is initially processed by a Weather Modeling module containing V ponderation layers and a smoothing layer as defined Section 6.3. Each of the V weighting layers, designated as v is associated with an MLP layer, enabling the I signals to be weighted into F_v signals, with $I >> F_v$. This enables the network to identify multiple potential weightings. The $F_w = \sum_{v=1}^{V} F_v$ weighted signals are then concatenated into a vector of size $B \times H \times F_w$. The F_T

signals corresponding to aggregated temperatures are smoothed by a smoothing layer being either a recurrent network (RNN, LSTM or GRU), or an exponential smoothing layer as defined in Section 6.3. They are then concatenated to the vector, now of size $B \times H \times (F_w + F_T)$. The set of hyperparameters for the Weather Modeling module is called λ_w and include the V output dimensions of the weighting layers for each of the V variables, as well as the type of layer used for smoothing along, with the hyperparameters associated with that layer.

Load Forecasting Network The vector generated by the Weather Modeling module is merged with the other vector of features, designated as c_b, resulting in a vector x_b of size $B \times H \times (F_w + F_T + B)$, which is then fed to the load forecasting model. This model is identical to the one used in Appendix C.4. The load forecasting network should map $x_b \in \mathbb{R}^{B \times H \times (F_w + F_T + F)}$ into the target $y_b \in \mathbb{R}^{B \times H}$. For this, two DAGs Γ_1 and Γ_2 are used. The graph Γ_1 is made of 2-dimensional layers operations to treat the 2-dimensional input and is parameterized by α_1 and λ_1. A flattened layer follows Γ_1 to transform the 2-dimensional latent representation into a 1-dimensional one. The graph Γ_2 is then made of 1-dimensional layers operations and is parameterized by α_2 and λ_2. We have $\alpha = [\alpha_1, \alpha_2]$ and $\lambda = [\lambda_1, \lambda_2]$. A final output layer maps the output shape of Γ_2 to H.

Finally, a Kalman adaptation is made using two last hyperparameters Q and σ. We call λ_K this hyperparameter set. To summarize, our search space can be written as: $(\alpha, \lambda) = (\{\alpha_1, \alpha_2\}, \{\lambda_w, \lambda_1, \lambda_2, \lambda_K\}) \in \big(\mathcal{A} \times \Lambda(\alpha)\big)$.

6.5 Experiments

In this section, we evaluate the efficacy of our weather modeling and Kalman adaptation techniques on the French load dataset from January 2023 to May 2024. In contrast with Chapter 5, which compares data from a relatively stable and distant period, our analysis focuses on a more dynamic and operational context. The training period spans from 2018 to 2022 and encompasses both the global pandemic caused by the SARS-CoV-2 virus and the subsequent energy crisis at the end of 2022. The test period encompasses the winter of 2023, during which consumers were encouraged to voluntarily reduce their consumption, a period commonly referred to as the "sobriety period".

6.5.1 Data

The load dataset was obtained from the website of the French Transmission System Operator (RTE)[2] and contains the French national load data at half-hourly intervals. Therefore, each day contains $H = 48$ time steps. The models were trained from January

[2] https://www.rte-france.com/eco2mix

2018 to December 2022 and subsequently evaluated from January 2023 to May 2024. The training was minimizing the training loss ℓ_{train}, which is the Mean Squared Error for those experiments. The objective funtion ℓ is the Mean Absolute Percentage Error (MAPE). Given a load series $Y = (\mathbf{y}_1 \dots \mathbf{y}_N)$ and the forecasts $\hat{Y} = (\hat{\mathbf{y}}_1, \dots \hat{\mathbf{y}}_N)$, $\text{MAPE}(Y, \hat{Y}) = 1/N \sum_{i=1}^{N} \left|(\mathbf{y}_i - \hat{\mathbf{y}}_i)/\mathbf{y}_i\right|$. The weather data set comprises three-hourly weather forecasts for 32 weather stations across France (see Figure 6.2). These forecasts are provided by Meteo France[3]. Prior to employing this data in our forecasting models, we performed a temporal linear interpolation. The other explanatory features used to explain the load data are calendar features including the day of the week, the month, the year, and whether the day in question fell on a public holiday or a surrounding day.

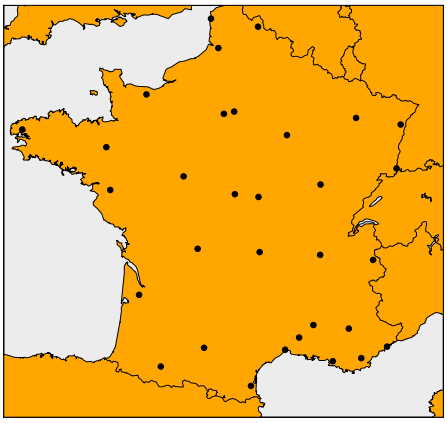


Figure 6.2: Location of the 32 french weather stations from our spatio-temporal weather dataset.

6.5.2 Baseline

We compare our results to models at the state-of-the-art in load forecasting: the day-ahead load forecast provided on RTE website, a Generalized Additive Model (GAM) used in the industry and EnergyDragon as proposed Appendix C.4.3. In the case of the GAM model, a single model is calibrated for each instant, resulting in a total of 48 models. The training set was modified by removing periods corresponding to lockdowns that were implemented during the pandemic caused by the novel coronavirus. EnergyDragon produces daily forecasts of $H = 48$ values, necessitating the use of a single model for all instants. A "Covid" feature has been incorporated into the model to indicate which periods corresponding to lockdowns are retained in the training set. With the exception of the "Covid" variable, the features are identical between the GAM and EnergyDragon models. The weather variables utilized in this study correspond to the weather at the 32

[3] https://www.data.gouv.fr/fr/organizations/meteo-france/

stations, with the data weighted and smoothed in accordance with the recommendations outlined in the RTE report on incorporating climatic contingencies into consumption forecasts[4]. Both models are recalibrated in an identical manner, utilising a Kalman filter that is updated on a daily basis with data from two days ago. To optimize Q and σ for the GAM model, an Iterative Grid Search was employed with the years 2020 to 2022 as validation set. For EnergyDragon, the years 2018 to 2020 were used as training dataset ($\mathscr{D}_{\text{train}}$) and the years 2021 and 2022 as validation set ($\mathscr{D}_{\text{valid}}$). Concerning the RTE model, no information is given on the structure of the model or its recalibration. For our model, later called ED Weather Modeling (for EnergyDragon Weather Modeling), which includes space-time weather modeling, we remove all weather-related features from EnergyDragon (all weather variables and their smoothed versions).

6.5.3 Results

Model	MAPE	Recalibration	MAPE
RTE	-	Not specified	2.316%
GAM	7.429	Kalman	2.019%
EnergyDragon	**2.988**	Kalman	1.947%
ED Weather Modeling	3.501	Kalman	**1.848%**

Table 6.1: Results over 2023 - May 2024

We evaluated each algorithm from the baseline on the French load signal. Both versions from EnergyDragon were run using 20 GPUs V100. The search algorithm used for the optimization is the steady-state evolutionary algorithm used in Chapter 5. The initial population is of size 100. Each algorithm was run with a global seed of 0 to ensure reproducibility. The results can be found in Table 6.1 and support the findings of Chapter 5. Indeed, even during an erratic period, EnergyDragon managed to beat the static version of GAM. As anticipated, GAM's Kalman recalibration is superior to EnergyDragon's static version, thereby validating the incorporation of a DNN recalibration brick. This addition, in combination with the optimization of σ and Q, have proven to be effective, as evidenced by the superior performance of the recalibrated EnergyDragon model in comparison to both RTE and GAM-Kalman. With regard to the incorporation of the weather modeling module, in the recalibrated version it enables us to achieve a slight improvement (5%) in mean absolute percentage error (MAPE) compared to EnergyDragon.

[4] https://www.services-rte.com/files/live/sites/services-rte/files/documentsLibrary/2022-04-01_REGLES_MA-RE_SECTION_2_F_3590_en

During the optimization phase, we noticed that the exponential smoothing layer exhibits superior performance in comparison to alternative modules. As Figure 6.3 shows, there are rapidly no RNNs left in the new DNNs created. We noticed they are no longer employed after the 200th neural network is created (the initial population is 100 individuals, and ultimately, more than 2.000 models are evaluated during optimization). An examination of the output of the weather modeling module reveals the manner in which the DNN has modeled the weather. This modeled weather can then be compared to the data supplied by RTE, which was used in the GAMs and EnergyDragon models. Figures 6.4 and 6.5 show a comparison between the data as modeled by the functions given by RTE, and the one found by two DNNs using the Weather Modeling modules and achieving good performance (respectively 2.1% and 1.85% of MAPE). We can see that DNN's modeling close from that proposed by RTE, without being identical. It's interesting to notice, for example, that wind is almost identically represented, while for temperature we have two different aggregations. One is very close to the signal proposed by RTE, the other is opposite and larger in amplitude. As for smoothing, while Figure 6.5 has a smoothing fairly close to that used in the GAM and EnergyDragon models, for Figure 6.4 the first smoothing coefficient is much lower.

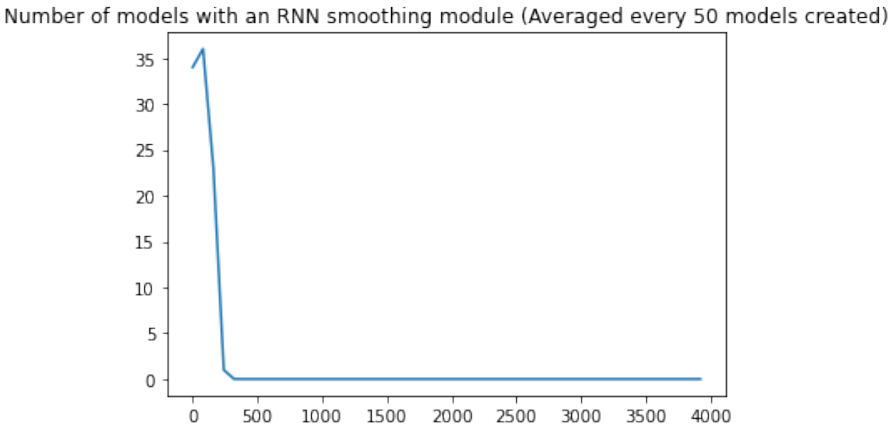
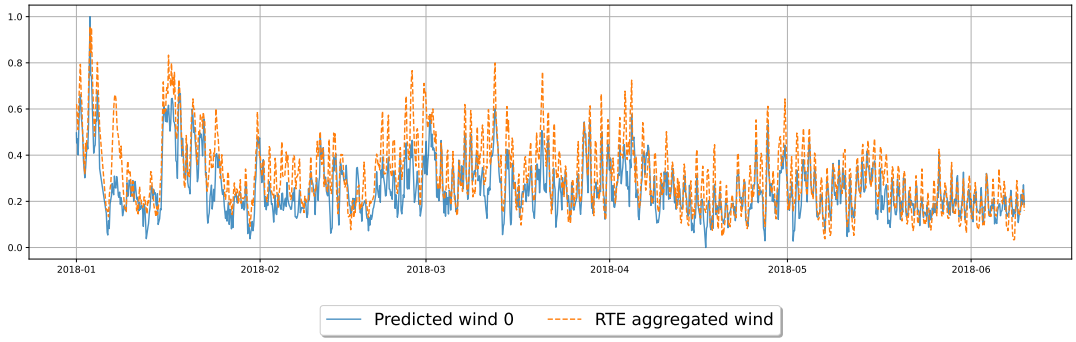
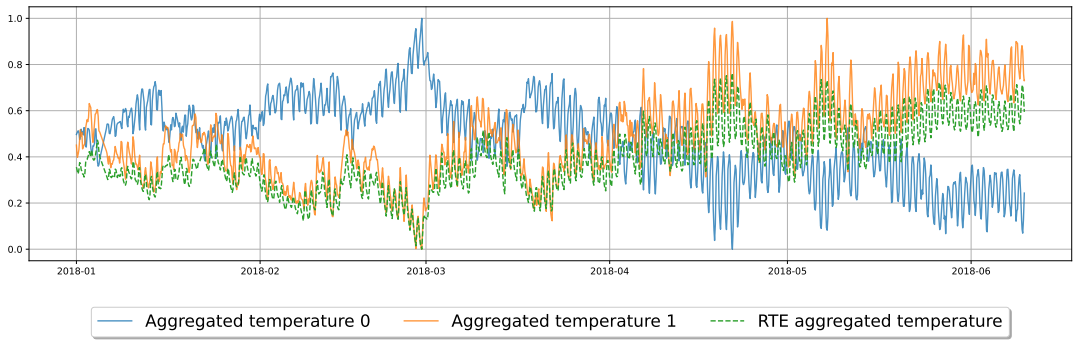


Figure 6.3: Number of DNNs created having a Recurrent Neural Network as smoothing layer through time.

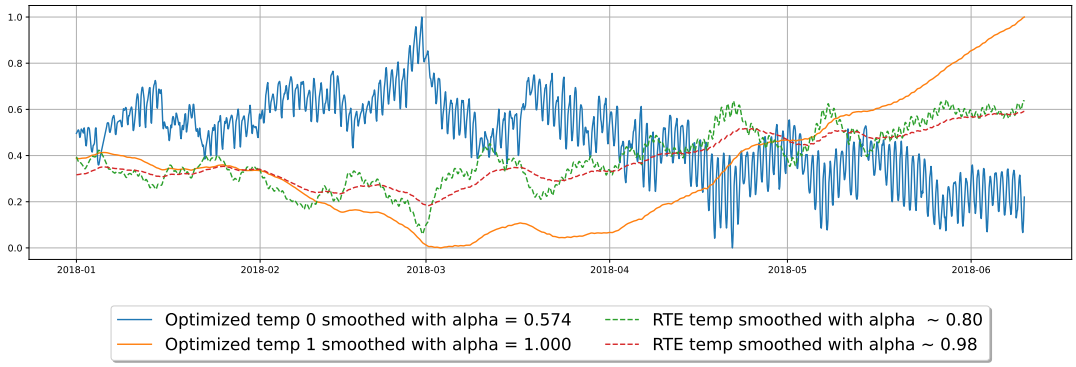
Finally Figure 6.6 shows the models found by EnergyDragon with and without the weather modeling part. We focused on the core of the network without showing the weather modeling brick, to only compare the load forecasting network. The structure of EnergyDragon with the weather modeling network shown Figure 6.6b is a lot simpler than the one without shown Figure 6.6a. It can be hypothesized that the weather is represented in a more comprehensible way for the DNN thanks to the weather modeling module. As a result, fewer transformations would be required to output the load forecast.



(a) Dotted line: the wind signal aggregated using weights from RTE, used in the GAM and EnergyDragon models. Solid line: the wind signal aggregated by the weather modeling module.

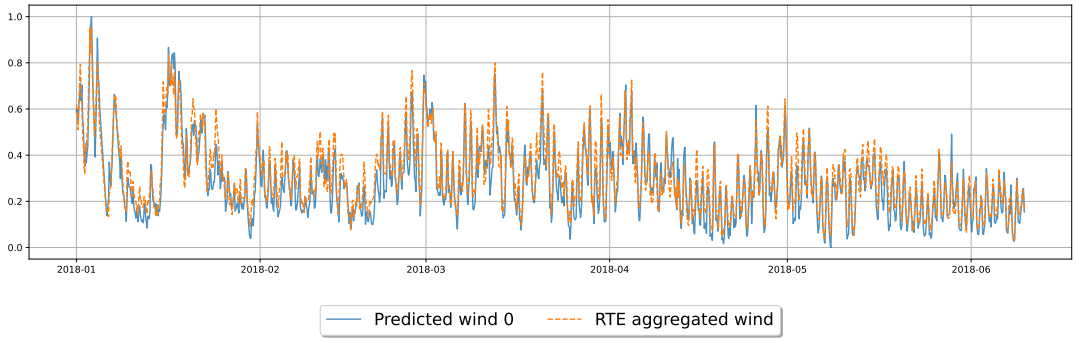


(b) Dotted line: the temperature signal aggregated using weights from RTE, used in the GAM and EnergyDragon models. Solid lines: two aggregated temperature signals found by the weather modeling module.

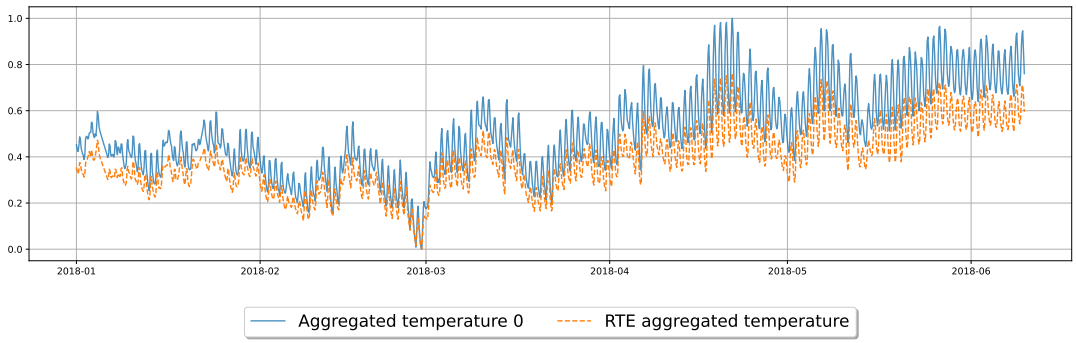


(c) Dotted line: the smoothed temperature signals used in the GAM and EnergyDragon models. Solid lines: the two aggregated temperature signals smoothed by the weather modeling module.

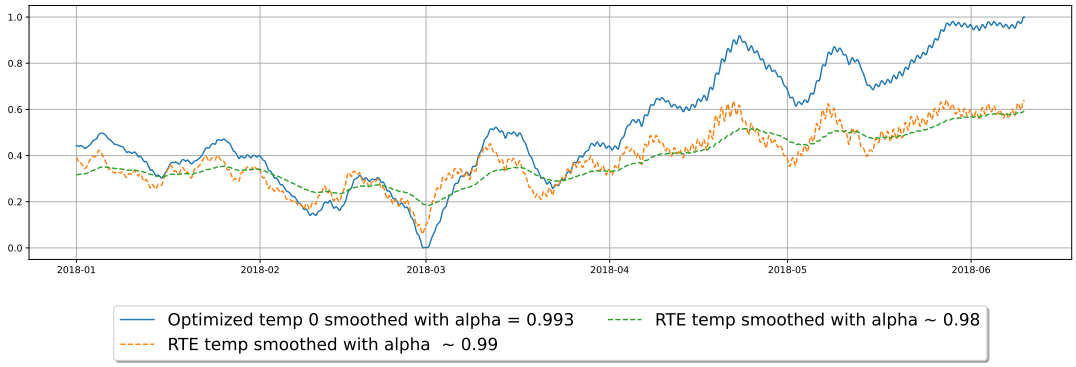
Figure 6.4: Comparison between the weather as modeled within the GAM and EnergyDragon models, versus the weather modeled by the DNN based Weather Modeling module, for a model having a MAPE of 2.1%



(a) Dotted line: the wind signal aggregated using weights from RTE, used in the GAM and EnergyDragon models. Solid line: the wind signal aggregated by the weather modeling module.

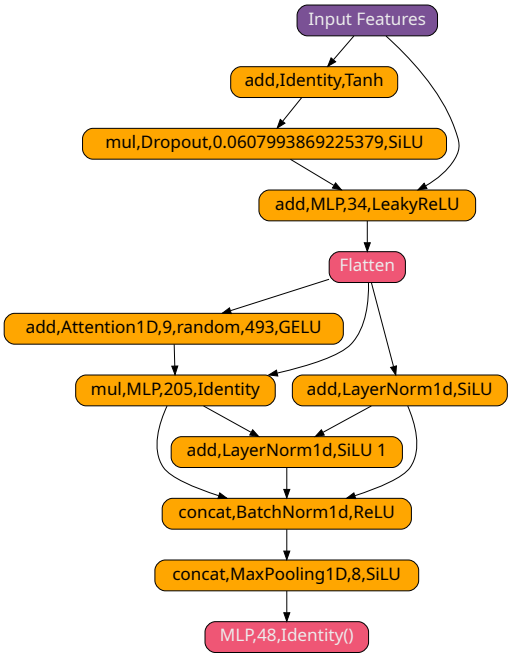


(b) Dotted line: the temperature signal aggregated using weights from RTE, used in the GAM and EnergyDragon models. Solid line: the aggregated temperature signal found by the weather modeling module.

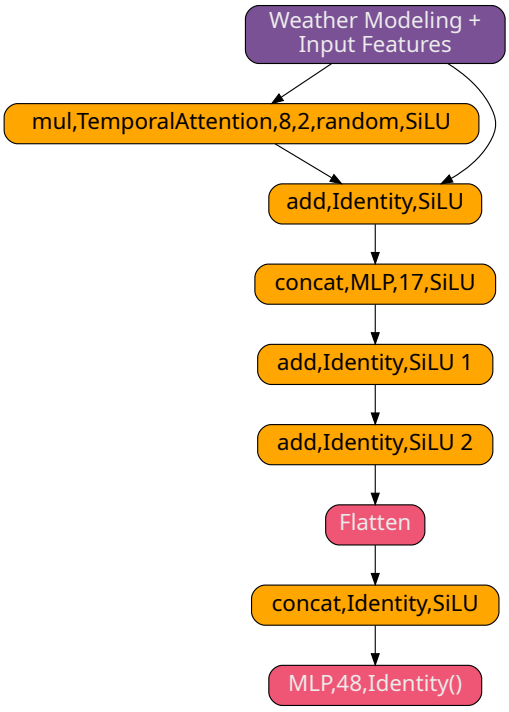


(c) Dotted line: the smoothed temperature signals used in the GAM and EnergyDragon models. Solid line: the aggregated temperature signal smoothed by the weather modeling modeling module.

Figure 6.5: Comparison between the weather as modeled within the GAM and EnergyDragon models, versus the weather modeled by the DNN based Weather Modeling module, for a model having a MAPE of 1.85%



(a) Best model found by EnergyDragon without the automated weather modeling part.



(b) Best model found by EnergyDragon with the automated weather modeling part.

Figure 6.6: Best models found by EnergyDragon without and with the automated weather modeling module.

6.6 Conclusion

Finally, this chapter builds on the work presented in Chapter 5 on automated deep learning for load forecasting. That chapter proposed a framework, called EnergyDragon, for optimizing the architecture and hyperparameters of deep neural networks specifically designed for load forecasting. This new chapter improves on the previous one by incorporating an automated spatio-temporal weather modeling approach based on DNN and a recalibration module based on Kalman filtering. The effectiveness of our approach is evaluated in a more dynamic and operational context, namely the French national load during the sobriety period.

To automate the spatio-temporal representation of the weather, we have maintained a close alignment with the functions used in the state of the art for load forecasting. In the Section 6.5, we show that the representations identified by our DNNs are close to those used in the other models from our baseline. This approach has the advantage of remaining interpretable, allowing a comparison between the DNN-generated model and the insights derived from domain expertise. However, these preliminary results could probably be further improved by incorporating more sophisticated layers of DNNs and taking advantage of over-parameterization.

Chapter 7

WindDragon: Automated Deep Learning for regional wind power forecasting

In this chapter, we present a second application of DRAGON to energy sustainability, namely regional wind power forecasting. Achieving net-zero carbon emissions by 2050 will require the integration of significant wind power capacity into national power grids. However, the inherent variability and uncertainty of wind power poses significant challenges to grid operators, particularly in maintaining system stability and balance. Accurate short-term forecasting of wind power is therefore essential. This chapter presents an innovative framework for regional-level wind power forecasting over short-term horizons (1 to 6 hours) using an automated deep learning regression framework called *WindDragon*, built with the tools of the DRAGON package introduced in Chapter 4. Specifically designed to process wind speed maps, *WindDragon* automatically builds deep learning models using Numerical Weather Prediction (NWP) data to provide state-of-the-art wind power forecasts. We perform extensive evaluations on data from France for the year 2020, benchmarking *WindDragon* against a variety of baseline frameworks, including both deep learning and traditional methods. The results show that *WindDragon* achieves significant improvements in forecast accuracy over the baseline approaches, highlighting its potential to improve grid reliability in the face of increased wind power integration.

Keisler, J. and Le Naour, E. **WindDragon: Automated Deep Learning for regional wind power forecasting.** *Submitted*, 2024.

Contents

7.1 Introduction

Global context To meet the 2050 net zero scenario envisaged by the Paris Agreement (United Nations Convention on Climate Change, 2015), wind power stands out as a critical energy source for the future. Remarkable progress has been made since 2010, when global electricity generation from wind power was 342 TWh, rising to 2,100 TWh in 2022 (International Energy Agency (IEA), 2023). The IEA targets approximately 7,400 TWh of wind-generated electricity by 2030 to meet the zero-emissions scenario. However, to realize the full potential of this intermittent energy source, accurate forecasts of wind power generation are needed to efficiently integrate it into the power grid.

Regional wind power forecasting Most of the work in the literature on wind power forecasting is done at a local scale, i.e. an individual wind farm or turbine. In this paper we focus on a more global scale, the aggregated production of a country or a large region. Regional wind power generation forecast is critical in the context of the European electricity market for several reasons. (i) First, a short-term forecast of up to 48 hours is useful for the spot (day-ahead) market, which sets the "final" price of electricity hour by hour according to supply and demand. (ii) Secondly, Short-term forecasts are useful for the TSO (Transmission System Operator), which has to ensure the balance between supply and demand on the transmission network within its perimeter. (iii) Finally, in the longer term, up to a few days, regional wind power forecasts can be used to anticipate downturns. They correspond to a situation in which a large amount of renewable energy is fed into the grid at the same time. Renewable energies indeed have market priority over, for example, nuclear or coal, which are more expensive to produce.

Wind power generation forecast at a global scale can be done in two ways, either by forecasting each farm in the region (or even each wind turbine) and then adding these forecasts together, or by directly forecasting the aggregated signal. The first method is impractical for the majority of operators, as it requires production data for each park, which is confidential. Moreover, even in cases where the data is available,

Wang et al. (2017) pointed out that having forecast system for each wind farm in the region considered can be too costly for some forecast service providers. In this paper we focus on wind power generation forecast at a global scale.

Contributions In this chapter, we propose to leverage the spatial information in NWP wind speed maps for national wind power forecasting by exploiting the capabilities of DL models. The overall methodology is illustrated in Figure 7.1. To fully exploit the potential of DL mechanisms, we introduce WindDragon, an automated deep learning framework that uses the tools in the DRAGON package introduced Chapter 2 and Chapter 4. WindDragon attempts to automatically design well-performing neural networks for short-term wind power forecasting using NWP wind speed maps. WindDragon's performance will be benchmarked against conventional computer vision models such as Convolutional Neural Networks (CNNs) as well as standard baselines in wind power forecasting. The contributions of this work can be summarized as follows:

- We develop a novel automated deep learning framework specifically tailored to forecast aggregated wind power generation from wind speed maps.
- The proposed framework, named WindDragon, is designed to fully leverage the spatial information embedded in wind speed maps and can accommodate increases in installed capacity, making it adaptable and reusable.
- We conduct extensive experiments that demonstrate that WindDragon, when combined with Numerical Weather Prediction (NWP) wind speed maps, significantly outperforms both traditional and state-of-the-art deep learning models in wind power forecasting.

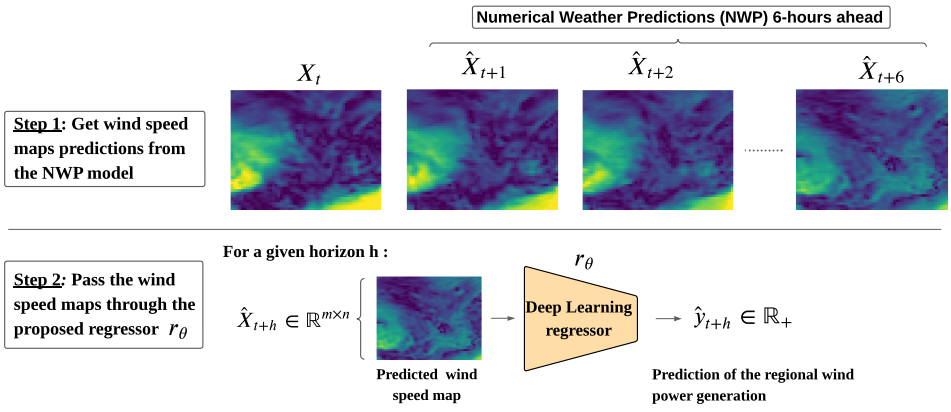


Figure 7.1: Global scheme for wind power forecasting. Every 6 hours, the NWP model produces hourly forecasts. Each map is processed independently by the regressor which maps the grid to the wind power corresponding to the same timestamp.

7.2 State-of-the-art

Wind power forecasting at the level of a single wind farm is a mature discipline (Jonkers et al., 2024) on forecast horizons ranging from the next minutes to the next days (see Kariniotakis 2017 for a book on the subject). However, the regional forecasting remains largely unexplored in the literature (Higashiyama et al., 2018).

7.2.1 Regional wind power forecasting

Transfer strategy Some studies have attempted to take advantage of the wealth of research at the turbine or wind farm scale to forecast regional wind energy. The general idea is to apply a forecasting model to wind turbines or farms whose data are available within the region and use a transfer function to move from local to regional data. For instance, Pinson et al. (2003) mentioned a model based on online persistence scaled with a ratio of the total installed capacity in the region and the capacity of wind farms for which online measures are available. Camal et al. (2024) forecasted the production of any wind farm in the control area of a TSO, taking into account the information collected from other wind farms. The method combines feature selection, regularization and local-learning via conditioning on recent production levels or on expected weather conditions.

Input dimension reduction Approaches that have attempted to forecast regional wind production directly from meteorological data such as NWP maps, or by incorporating operational variables from the (potentially numerous) wind farms in the region, have quickly run into the problem of the large size of the input data. Camal et al. (2024) noticed that at the scale of a region or of a country, the number of explanatory variables grows linearly with the number of explanatory sites or the number of variables considered per site. Both statistical and Machine Learning models face in this case the curse of dimensionality. Therefore, regularization or feature selection was investigated to mitigate the high dimensionality of the input features. Siebert (2008) used a clustering algorithm based on k-means and a mutual information-based feature selection algorithm to determine the best set of features for the forecast model. Lobo and Sanchez (2012) searched for samples with similar weather conditions. Davò et al. (2016) leveraged the principal component analysis (PCA) method to reduce the dimension of the data sets when forecasting regional wind power and solar irradiance. Wang et al. (2017) reduced the dimension of the NWP grid with the selection of minimum redundancy characteristics (mRMR) and PCA. They then applied a weighted average learning strategy to forecast the production of a Chinese region. In the work from Wang et al. (2018b), the spatio-temporal weather data is represented using a distance-weighted kernel density estimation model (DWKDE) which is the basis for a feature selection method based on mRMR. Finally, Wang et al. (2019c) performed probabilistic forecast with regular vine

copulad to reduce the weather dataset.

Although this input reduction is necessary for most Machine Learning models, deep learning models have demonstrated high capacities for extracting complex features from high-dimensional data.

7.2.2 Deep Learning for wind power forecasting

Deep learning models have been highly investigated for wind power forecasting both at the turbine level and at the regional aggregation level. A large variety of architectures have been used, depending on the input data available and the features that are sought to be extracted.

Yu et al. (2021) recognized the abilities of the deep learning model for non-linear mapping and massive data handling and used a feedforward neural network based on historical wind power and NWP information for regional wind power forecasting. To model the time dependencies of the wind power time series, many worked leveraged recurrent neural networks and their variants (long short-term memory or gated recurrent unit) such as Liu et al. (2021a) or Alkabbani et al. (2023). The interactions between several wind farms have been investigated using the Transformer model by Lima et al. (2022) and using graph neural networks by Qiu et al. (2024). The direct use of DNN directly on wind speed map has been tackled using convolutional neural networks (CNNs) which had shown strong capabilities for extracting relevant features from image data. Higashiyama et al. (2018) used 3-dimensional CNNs to forecast the production of a single wind farm based on NWP grids. Bosma and Nazari (2022) and Jonkers et al. (2024) proposed day-ahead regional wind power forecasting CNNs which architecture was inspired by Computer Vision models such as ResNet (see He et al. (2016)).

The challenge of wind power forecasting is that it combines dependencies to weather variables but remains a times series. Therefore architectures mixing various types of layers have been investigated to capture various dependencies. Miele et al. (2023) compared the performance of CNN-LSTM with a multi-modal neural network with two branches: one for the NWP grid and one for past data, for a single wind farm. Zhou and Lu (2023) combined convolution, LSTM and attention layers to forecast the production of a wind farm. Given this large variety of possible architectures, ones might want to use automated tools to find the best one for the dataset at hand.

7.2.3 Automated Deep Learning

Main concepts The research field related to the automation of deep neural network design is called Automated Deep Learning (AutoDL). It belongs to a more global research area called Automated Machine Learning (AutoML) which studies the automatic design of high-performance Machine Learning models. As any AutoML approaches, AutoDL systems consist of three main components: the *search space*, the *search strategy*

and the *performance evaluation*. The *search space* should contain all the considered neural networks architectures and hyperparameters which is the set of all available design choices, like the number and type of layers in the neural network, the connection between the layers or the training parameters, like the learning rate. The *search strategy* will determine how to navigate within the search space to select promising configurations. The bigger the search space, the more sophisticated the search strategy should be for effective exploration. The *performance evaluation* will assess the performance of the candidate configurations until the search strategy finds a suitable neural network (usually best configuration found after a given number of evaluations).

AutoDL for wind power forecasting A few works have applied AutoDL to wind power forecasting, such as Tu et al. (2022) or Jalali et al. (2022). However, these approaches are limited to optimizing the hyperparameters of one type of architecture, possibly integrating a few architectural hyperparameters such as the number of layers. The AutoDL community has developed a large number of tools to optimize neural network architectures more broadly, but as Tu et al. (2022) points out, the search spaces used by these approaches are tailored to Computer Vision and Natural Language Processing tasks. For example, Hutter et al. (2019) reviewed many approaches based on (hierarchical) cell-based search spaces, where the neural networks are represented as a sequence of small iterated Directed Acyclic Graphs (DAGs) called cells. The architecture of the cell is optimized and then the pattern is repeated throughout the network. Such an approach is efficient for Computer Vision tasks, where models that repeat sequences of convolutional pooling layers and skip connections are very powerful. Another popular approach is DARTS, proposed by Liu et al. (2018d), which uses a meta-architecture that is designed to include all possible architectures. The general structure of the network is fixed, and for each layer several candidate operations are possible. Each is associated with a probability of being chosen, which is optimized by gradient descent. This approach, which is effective for generating architectures based on 3×3 or 5×5 convolutions, has a very limited search space and assumes that the subgraph obtained by keeping only the operation with the highest probability for each layer is the optimal graph. More diverse tasks have been tackled by the AutoDL framework AutoPytorch, which offers a version for tabular data, described in Zimmer et al. (2021), and for time series forecasting, see Deng et al. (2022), providing search spaces of MLPs and residual connections for the tabular version, and various encoder/decoder blocks for the time series version to cover several state-of-the-art architectures in time series (e.g: TFT from Lim et al. 2021, NBEATS from Oreshkin et al. 2020, or DeepAR from Salinas et al. 2020). All search spaces for the above AutoDL approaches have been restricted to allow effective searching. This observation is shared more generally by recent reviews such as White et al. (2023) on AutoDL and Baratchi et al. (2024) on AutoML. In the case of wind production forecasting, as indicated by Tu et al. (2022), we would like to have a search space for designing architectures that combine different types of layers such as

MLPs, CNNs, or attention, that also have computational graphs that are more complex than a linearly sequential architecture, and whose hyperparameters can be optimized, as they are crucial in this type of task. The AutoDL package DRAGON, introduced in Chapter 2, provides tools for designing such search spaces.

7.2.4 DRAGON package

DRAGON, or DiRected Acyclic Graphs optimizatioN, is an open-source Python package[1] offering tools to conceive Automated Deep Learning frameworks for diverse tasks. The package is based on three main elements: building bricks for search space design, search operators for those bricks and search algorithms.

Search space DRAGON offers several building bricks to encode deep neural networks architectures and hyperparameters. The network structures are represented as Directed Acyclic Graphs, where the nodes represent the layers and the edges the connection between them. The layers are encoded by a succession of three elements: a combiner, an operation and an activation function. As no constraint is made on the graph structure, each node may receive an arbitrary number of incoming inputs of various size. They are gathered into a single input through the combiner. The operation can be any *PyTorch* building block parametrized by a set of hyperparameters. The DRAGON user has to specify which kind of building blocks the search space should contain, and for each, the associated hyperparameters. Besides the DAGs, the user can chose to optimize other hyperparameters such as the learning rate, the output shape of the last layer, etc. The hyperparameters may be numerical or categorical. The graph encoding can be used to represent the entire structure, but it is also possible to design more specific search spaces for certain applications. For example, it is possible to combine different graphs for a Transformer-type structure (see Vaswani et al. 2017 for an introduction to the Transformer model), with one graph for the encoder part and another graph for the decoder part, in order to impose a two-part structure.

Performance evaluation The search space is designed for a specific performance evaluation strategy, which will assess the score of a given configuration from the search space. In the case of DRAGON, the user has to define its own performance evaluation. Given an element from the search space, the performance evaluation should at least build a *PyTorch* model and perform any type of training/validation process on the data.

Search Operators Each building blocks from DRAGON comes with a *neighbor* attribute that defines how to create a neighboring value from a representation. Those operators can be seen as mutations in the case of an evolutionary algorithm or neighborhood operators for a simulated annealing or a local search. In the case of an integer

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

for example, the *neighbor* attribute will pick the new value in a range surrounding the actual one. For the DAGs, it is possible to add or delete nodes, or to modify the edges and the nodes contents.

Search Algorithms The package implements several search strategies which may use the search operators and can be distributed in a high performance computing (HPC) environment. Besides the random search, Hyperband (see Li et al. (2018)), an evolutionary algorithm and Mutant-UCB presented in Chapter 3 are available. They take as input the search space and the performance evaluation designed by the user and return the best configuration.

7.3 WindDragon

We used the tools provided by DRAGON to create WindDragon, an AutoDL framework for regression on wind speed maps towards regional wind power forecasting. The framework takes as input two datasets $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{valid}}$. Each dataset $\mathcal{D}$ is made up of pairs (X_t, Y_t) for several time steps t, where $X_t \in \mathbb{R}^2$ is a wind speed map and $Y_t \in \mathbb{R}^R$ are the associated wind production values, one for each of the R regions. The framework is first creating wind speed maps by region r: X_t^r. Two datasets $\mathcal{D}^r_{\text{train}} = (X^r, Y^r)$ and $\mathcal{D}^r_{\text{valid}} = (X^r, Y^r)$ are put together for each region r with these regional wind speed maps and the associated regional production. WindDragon aims at finding, for each region r, the optimal model $\hat{f}^r$ from a search space Ω with respect to a loss function ℓ such that:

$$\hat{f}^r \in \underset{f \in\ \Omega}{\operatorname{argmin}}\, \ell\big(f_{\hat{\delta}}, \mathcal{D}^r_{\text{valid}}\big), \tag{7.1}$$

where the model $f_{\hat{\delta}}$ corresponds to the model $f \in \Omega$ trained on $\mathcal{D}^r_{\text{train}}$.

7.3.1 Search space and performance evaluation

Data processing The input data X_t contains the wind speed map corresponding to the whole country and has to be divided into regional data. As shows Figure 7.2 for a specific region (here Auvergne-Rhône-Alpes), wind turbines are not evenly distributed across the administrative regions. Therefore, instead of using them, we draw areas around each wind farm in the region and took the convex hull of all the considered points. The result is a seamless map $X_t^r \subset X_t \in \mathbb{R}^2$ that includes local wind turbines with no gaps to disrupt the models. The areas surroundings the wind farms are drawn according to a distance parametrized by a parameter called $g \in \mathbb{N}^\star$. When g gets higher, the convex hull becomes larger. Installed capacity data - corresponding to the maximum wind power a region can produce - for each region and each time step t is available and updated every three months. It was collected and used to scale the wind power target

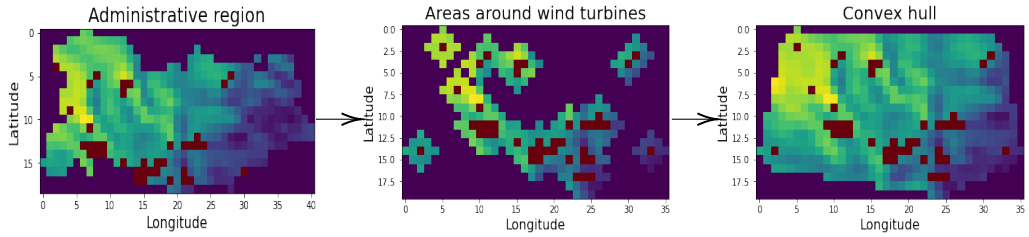


Figure 7.2: Data preparation for the region Auvergne-Rhône-Alpes. The wind farms are represented in red. The first image shows the distribution of wind farms across the administrative region.

to train the models. Training the model f on the region r with respect to the training loss ℓ_{train}, means finding the model optimal weights $\hat{\delta} \in \Delta$ such that:

$$\hat{\delta} \in \underset{\delta \in \Delta}{\operatorname{argmin}}\, \ell_{\text{train}}\big(f_\delta(X^r), \frac{Y^r}{c^r}\big), \tag{7.2}$$

where $c^r \in \mathbb{R}$ is the installed capacities for the region r and $\mathcal{D}^r_{\text{train}} = (X^r, Y^r)$. The evaluation of the model f on $\mathcal{D}^r_{\text{valid}}$ is made on the de-normalized value of Y^r.

Search space Each model $f \in \Omega$ has to forecast a one-dimensional output $Y^r_t \in \mathbb{R}$ from a two-dimensional input: the wind speed map $X^r_t \in \mathbb{R}^2$. Therefore, each neural network from Ω is made of two Directed Acyclic Graphs as represented in Figure 7.3. A first graph Γ_1 processes two-dimensional data and can be composed by convolutions, pooling, normalization, dropout, and attention layers. Then, a flatten layer and a second graph Γ_2 follow. This one is composed by MLPs, self-attention, convolutions and pooling layers able to treat one-dimensional. A final MLP layer is added at the end of the model to convert the latent vector to the desired output format. The detailed operations and hyperparameters available within WindDragon are detailed in Table 7.1. Regarding the parameters external to the architecture, the weather map size parameter g is also optimized. The search space is then: $[\Gamma_1, \Gamma_2, o, g]$ where o represents the final MLP layer, which is a constant.

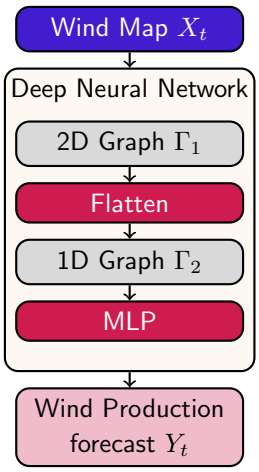


Figure 7.3: WindDragon's meta model for wind power forecasting

Performance Evaluation The performance evaluation takes as input a region r and a configuration from the search space and will:

- Construct the datasets $\mathcal{D}^r_{\text{train}}$ and $\mathcal{D}^r_{\text{valid}}$ from $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{valid}}$ according to the parameter g parameterizing the grid size, from the configuration.
- Build the model f with the elements from the configuration and train the model on $\mathcal{D}^r_{\text{train}}$ according to Equation (7.2).
- Evaluate the performance of $f_{\hat{\delta}}$ on $\mathcal{D}^r_{\text{valid}}$ according to Equation (7.1).

Table 7.1: Layers available and their associated hyperparameters in the WindDragon search space (for the first and the second graph).

Layer type	**Graph concerned**	**Optimized hyperparameters**	
Identity	Both	-	
Fully-Connected (MLP)	Both	Output shape	Integer
Self-Attention	Both	Initialization type Heads number Output dimension	[convolution, random] Integer Integer
1D Convolution	1D Graph Γ_2	Kernel size Output dimension	Integer Integer
2D Convolution	2D Graph Γ_1	Kernel size Output dimension	Integer Integer
1D Pooling	1D Graph Γ_2	Pooling size Pooling type	Integer [Max, Average]
2D Pooling	2D Graph Γ_1	Pooling size Pooling type	Integer [Max, Average]
1D Normalization	1D Graph Γ_2	Normalization type	[Batch, Layer]
2D Normalization	2D Graph Γ_1	Normalization type	[Batch, Layer]
Dropout	Both	Dropout rate	Float

7.3.2 Search Algorithm

Regarding the search algorithm, four are available within DRAGON: the Random Search, HyperBand (Li et al., 2018), an Evolutionary Algorithm, and Mutant-UCB. In Chapter 3 introducing this last algorithm, the four are compared and Mutant-UCB appears as the most efficient one.

Mutant-UCB This algorithm combines a multi-armed bandits approach with evolutionary operators. Each model $f \in \Omega$ corresponds to an arm, and choosing an arm corresponds to a partial training of the model. Indeed, training a neural networks takes a lot of time, and a lot of algorithms such as the Random Search or the Evolutionary Algorithms give the same amount of resources for all the evaluated configurations. It means such algorithms are loosing a lot of time and computational resources on bad configurations. Resource allocation strategies used for example by HyperBand, allows to gradually attribute ressources to the most promising solutions. A partial training can then be for example a training on a small set of data or with a small number of epochs. In short, Mutant-UCB generates a population of $K \in \mathbb{N}^\star$ of random configurations. For each arm k from this population, a partial training is made to get a first loss ℓ_k. Then, at each iteration i, an arm I_i from the population is drawn following an Upper-Confidence-Bound strategy:

$$I_i \in \underset{k\in\{1,\dots K\}}{\operatorname{argmin}} \left\{ \hat{\ell}_k - \sqrt{\frac{E}{N_k}} \right\},$$

where: $\hat{\ell}_k$ is the average loss for all the previous partial trainings of the model associated to the arm k, E is the exploration parameters and N_k the number of time the arm k has been picked. Once the arm I_i is chosen, with a probability $1 - \bar{N}_{I_i}/N$, the model is mutated, otherwise, a new partial training is done. The value N corresponds to the maximum number of partial trainings a model can have (to prevent overfitting) and $\bar{N}_{I_i}$ corresponds to the number of time the model associated to I_i has been trained. In the case of a mutant creation, the number of arms K increases and the new model is partially trained for the first time. For more information on Mutant-UCB please refer to Chapter 3.

Partial trainings In the original paper, the partial trainings were done on a small number of epochs. For WindDragon, we changed it to be a small number of epochs on a given region. Instead of running one version of Mutant-UCB, we ran Mutant-UCB for all the regions. We indeed make the assumption that a similar architecture will fit for all the regions, even if some layers or hyperparameters might change from one region to another. The input X^r might be of different shapes for different regions. This shape change is handle by DRAGON when building the neural network f. The layers and DAGs from the package may be adapted by weights cropping or padding to any new shape during the network initialization. Splitting the training between different regions follows the spirit of Mutant-UCB, where the loss minimized to pick the future arm relies on the empirical mean of the various partial trainings of a model f. The performance across the regions might be different, and converging towards a model generally good over all regions can be done by taking this empirical mean. To reduce the variance between the performance of the region, the loss ℓ considered to evaluate a model f on a given region

would be an error function (such as the mean squared error, the mean absolute error or a variant) of f, divided by this same error function but of a reference model. See Section 7.4 for more information.

7.4 Experiments

Datasets The wind speed maps used are 100-meter high forecasts at a 9 km resolution provided by the HRES[2] model from the European Centre for Medium-Range Weather Forecasts (ECMWF). The maps are provided at an hourly time step and there are 4 forecast runs per day (every 6 hours). Only the six more recent forecasts are used here as the forecasting horizon of interest is six hours. The hourly french regional and national wind power generation data as well as the french TSO hourly forecasts and the installed capacities values come from the ENTSOE-E Transparency Platform[3].

Baselines We use the following baselines to compare hourly forecasts for a horizon h ($h \in \{1, ..., 6\}$):

- **Persistence**: Given access to forecasts every 6 hours derived from the ground truth situation, the wind power value is also available at the same interval. Persistence involves replicating this value for the subsequent six hours. Therefore, the model predicts wind power generation at future time $t + h$ as equal to the observed generation at current time t.
- **XGB on Wind Speed Mean**: Forecasts wind power at $t + h$ using a two-step approach as depicted Figure 7.4: (i) Compute the mean wind speed for the considered region at $t + h$ using NWP forecasts. (ii) Apply an XGBoost regressor (Chen and Guestrin, 2016) to predict power generation based on the computed mean wind speed.

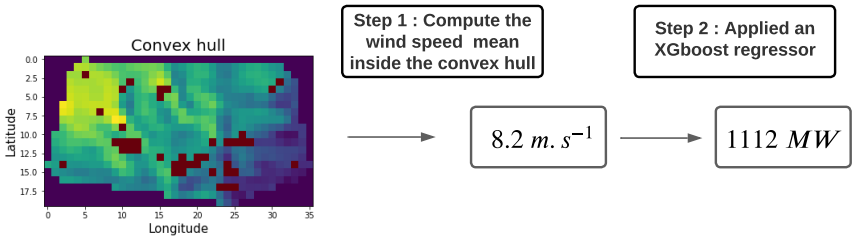


Figure 7.4: Visual illustration of the XGB two-steps approach on the Auvergne-Rhône-Alpes region.

[2] https://www.ecmwf.int/en/forecasts/datasets/set-i
[3] https://transparency.entsoe.eu/

- **Convolutional Neural Networks (CNNs)**. Use the same training setup as WindDragon: forecasts wind power at $t+h$ using the NWP forecasted wind speed map. CNNs can efficiently regress a structured map on a numerical value by learning local and spatial patterns (LeCun et al., 1995). In addition, the weight sharing induced by the convolutional mechanism reduces the number of learned weights compared to alternative deep learning mechanisms like dense (Haykin, 1994) or self-attention layers (Vaswani et al., 2017). This feature makes CNNs particularly effective when dealing with relatively small amounts of data. Figure 7.5 shows the architecture of the CNN baseline we implemented. We used a simple grid search to optimize the hyperparameters (e.g. the number of layers, the kernel sizes, the activation functions).

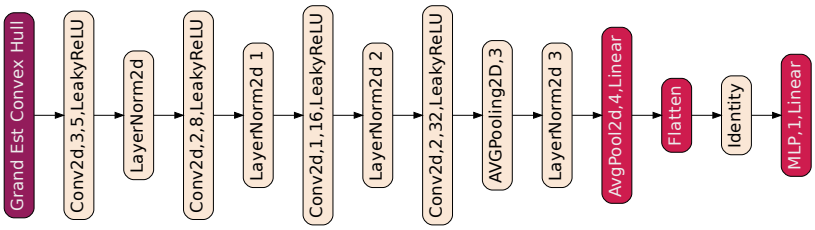


Figure 7.5: CNN architecture applied on the Grand Est region.

- **French TSO (RTE).** The european TSOs have to provide *Current*, *IntraDay* and *Day-Ahead* wind and solar forecasts. We have used the *Current* forecast within our baseline to put the results into perspective with operational values. The forecasting methods and horizons are not detailed. The regulatory article[4] only states that the published "Current" forecast is the latest update of the forecast. The information is regularly updated and published during intra-day trading. It is the closest setup of our experiments.

Experimental setup We used the years from 2018 to 2019 to train the models, and the data from 2020 is used to evaluate how the models perform. All the neural networks were trained using the Adam optimizer. The CNN was trained for 200 epochs. Mutant-UCB was run for 72 hours, with $N = 10$, $K = 600$, $E = 0.01$ and 20 epochs by partial training. The CNN model was given as input to the search algorithm. Among the first K models initialized, 10 had the CNN architecture, with values of g ranging from 1 to 10. The CNN losses were used to scale the regional errors for WindDragon. Mutant-UCB was distributed over 20 V100 GPUs and ran for 72 hours.

[4] https://transparencyplatform.zendesk.com/hc/en-us/articles/16648445340180-Generation-Forecasts-for-Wind-and-Solar-14-1-D

Results We computed two scores: **Mean Absolute Error (MAE)** in Megawatts (MW), showing the absolute difference between ground truth and forecast, and **Normalized Mean Absolute Error (NMAE)**, a percentage obtained by dividing the MAE by the average wind power generation for the test year. The MAE gives an idea of the amount of energy contained in the errors, while the NMAE enables performance to be compared between regions. We run experiments for each of the 12 French metropolitan regions and then aggregate the forecasts to derive national results. The national forecasting results are presented in Table 7.2, while detailed regional results can be found in Table 7.3.

Table 7.2: National results: sum of the regional forecasts for each models. The best results are highlighted in bold and the best second results are underlined.

	WindDragon		RTE		CNN		XGB on mean		Persistence	
	MAE (MW)	NMAE	MAE (MW)	NMAE	MAE (MW)	NMAE	MAE (MW)	NMAE	MAE (MW)	NMAE
France	**300.0**	**6.6 %**	482.1	10.6%	369.0	8.1 %	416.7	9.2 %	779.7	17.3 %

Table 7.3: Regional results. The best results are highlighted in bold and the best second results are underlined.

	WindDragon		CNN		XGB on mean		Persistence	
Region	MAE (MW)	NMAE	MAE (MW)	NMAE	MAE (MW)	NMAE	MAE (MW)	NMAE
Auvergne-Rhône-Alpes	**19.3**	**14.8 %**	19.6	15.0 %	29.2	22.4 %	28.7	22.0 %
Bourgogne-Franche-Comté	**30.0**	**13.6 %**	34.1	15.4 %	42.3	19.1 %	58.7	26.6 %
Bretagne	**33.7**	**13.2 %**	38.0	14.9 %	47.1	18.4 %	67.2	26.3 %
Centre-Val de Loire	**50.5**	**14.2 %**	57.3	16.1 %	61.9	17.5 %	96.7	27.3 %
Grand Est	**108.2**	**10.8 %**	130.5	13.1 %	148.8	14.9 %	251.2	25.1 %
Hauts-de-France	**140.7**	**10.6 %**	167.6	12.7 %	178.8	13.5 %	320.1	24.2 %
Île-de-France	**6.2**	**20.5 %**	7.2	23.7 %	7.5	24.9 %	9.5	31.5 %
Normandie	**27.4**	**11.8 %**	30.8	13.2 %	36.8	15.8 %	55.9	24.0 %
Nouvelle-Aquitaine	**37.8**	**13.8 %**	44.0	16.4 %	53.7	19.6 %	77.9	28.4 %
Occitanie	**51.1**	**12.3 %**	55.8	13.5 %	91.6	22.1 %	96.3	23.2 %
PACA	**3.2**	**29.7 %**	3.5	32.4%	4.5	41.4 %	4.3	39.5 %
Pays de la Loire	**34.1**	**12.5 %**	39.0	14.3 %	41.9	15.4 %	74.9	27.5 %

The results in Table 7.2 highlight three key findings:

(i) **Improved performance with aggregated NWP statistics.** Using the average of NWP-predicted wind speed maps coupled with an XGB regressor significantly outperforms the naive persistence baseline. It shows that the signal is closer to a regression problem than to a time series forecasting one. It is also interesting

to note that this simple model is already better than the signal produced by the French TSO.

(ii) **Gains from full NWP map utilization**. More complex patterns can be captured by using the full predicted wind speed map, as opposed to just the average, thereby improving forecast accuracy. In this context, the CNN regressor applied to full maps yielded gains of 47 MW (11.5%) over the mean-based XGB.

(iii) **WindDragon's superior performances**. WindDragon outperforms all baselines, showing an improvement of 69 MW (19%) over the CNN. On an annual basis, this corresponds to approximately 600 GWh. The average French citizen consumes between 2,500 and 3,000 kWh[5] of electricity per year. Therefore, 600 GWh per year is equivalent to the consumption of around 200,000 French inhabitants. The results underscore WindDragon's effectiveness in autonomously discovering the optimal deep-learning configurations for wind power regression. Moreover, Table 7.3 indicates the improvement is effective across all regions. During the optimization, WindDragon managed to find, for each region, a model outperforming each one from the baseline. The found architectures vary a bit from one region to another. Examples of the models output by WindDragon for various regions can be found Figures D.1, D.2, D.3, D.4 and D.5. The architectures mix various layers such as convolutions, pooling and normalization layers. The structures are very similar to a large two-dimensional graph, efficiently extracting spatial information from the input wind speed map and a small one-dimensional graph. The hyperparameters are however unique for each model.

Forecasts comparison In Figure 7.6, we present the aggregated national wind power forecasts using both WindDragon and the CNN baseline during a given week. While both models deliver highly accurate forecasts, it's important to highlight that WindDragon demonstrates superior accuracy, particularly during the high production level at the end of the signal. Figure D.6 show visual comparisons of all baseline performances on this same week. It appears that the models perform well at different times. For example, the RTE forecast is best for the small production spike in the middle of the day on 11 January, but much less good for the production dip on the night of 10 January. These differences in performance open the way to mixtures of models to further improve forecasts.

Performance analysis We compared the performance of the two best baselines, CNN and WindDragon, in more detail. Figure 7.7 shows the absolute errors and the normalised absolute errors by hour of the day and by month. In general, WindDragon is significantly better than CNN at all times of the day and in all months. On Figures 7.7a and 7.7b,

[5] based on the average European per capita consumption (Statista Research Department, 2022)

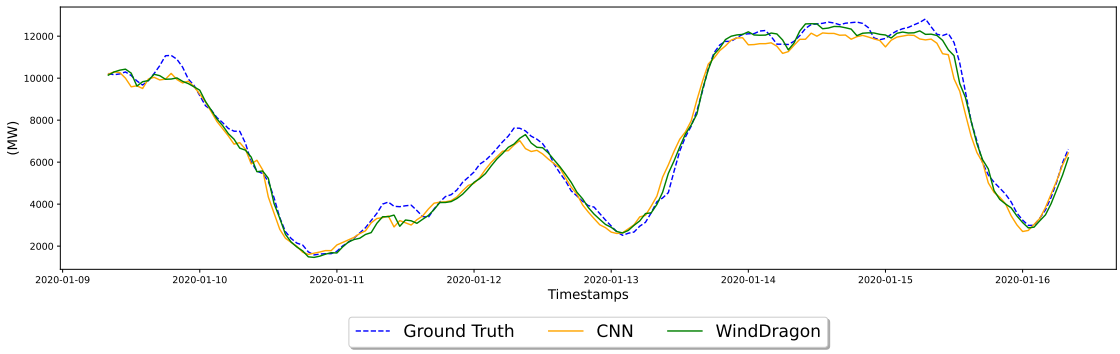
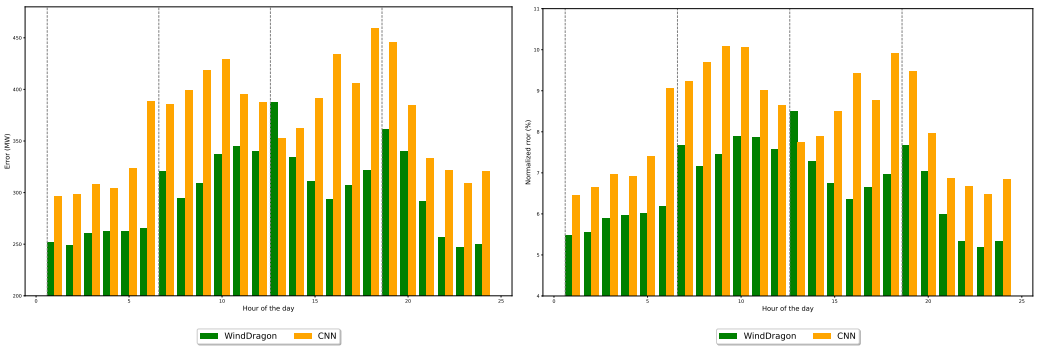


Figure 7.6: Wind power forecasts for a week in January 2020. The figure displays the ground truth as dotted lines, and the forecasts from the two top-performing models, WindDragon and the CNN.

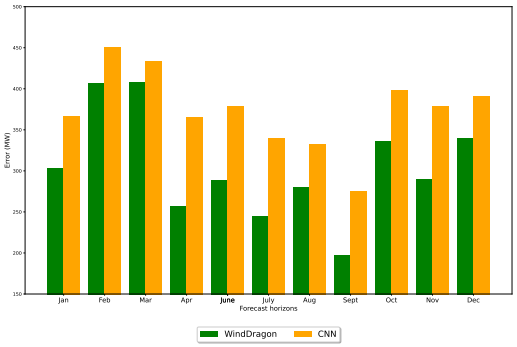
the dotted line represents the hour when a new NWP forecast arrives (every 6 hours). For the first two forecasts of the day (at midnight and 6 am), the performance of both models decreases as the forecast horizon increases. This is much more marked in the case of CNN, whose performance deteriorates dramatically, particularly at 6 am (when the forecast horizon is therefore 6 hours). This observation is less true for the later hours of the day. As for the months, the differences are more pronounced in summer, when wind power production is lower. Finally, we have plotted Figure 7.8 the mean absolute errors of CNN and WindDragon per quantile of the wind power distribution. We can see from this distribution that the two curves diverge particularly at the first quantile, where the production values are extremely low, and at the last quantile, where they are extremely high. The two curves never cross, demonstrating the homogeneous superiority of WindDragon over CNN.

Mutant-UCB time convergence Mutant-UCB ran for 72 hours on 20 GPUs. However, we saved the values of the models found by the algorithm as it ran so that we could analyse its convergence time. Figure 7.9a shows the best NMAE found per time step for each region. We can see that the performance converges very quickly during the first 2 hours of the algorithm before stabilising. Only a few regions such as Ile-de-France, Auvergne-Rhône-Alpes and Centre-Val de Loire show improvements in the last hours. Figure 7.9b zooms in on the first three hours of the algorithm. Except for PACA and Ile-de-France, most regions fall below 15% of NMAE in about an hour. Thus, although Mutant-UCB would have to be run for a long time to achieve very good performance, it is possible to obtain correct models in just one hour.

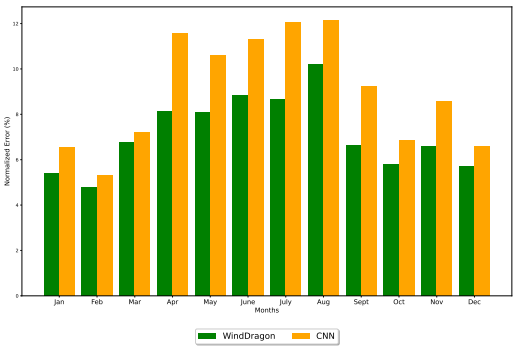


(a) Hourly error.

(b) Normalized hourly error



(c) Monthly error



(d) Monthly normalized error

Figure 7.7: Errors comparison between WindDragon and the CNN. The dotted vertical lines on Figures 7.7a and 7.7b represent the beginning of the new NWP forecast.

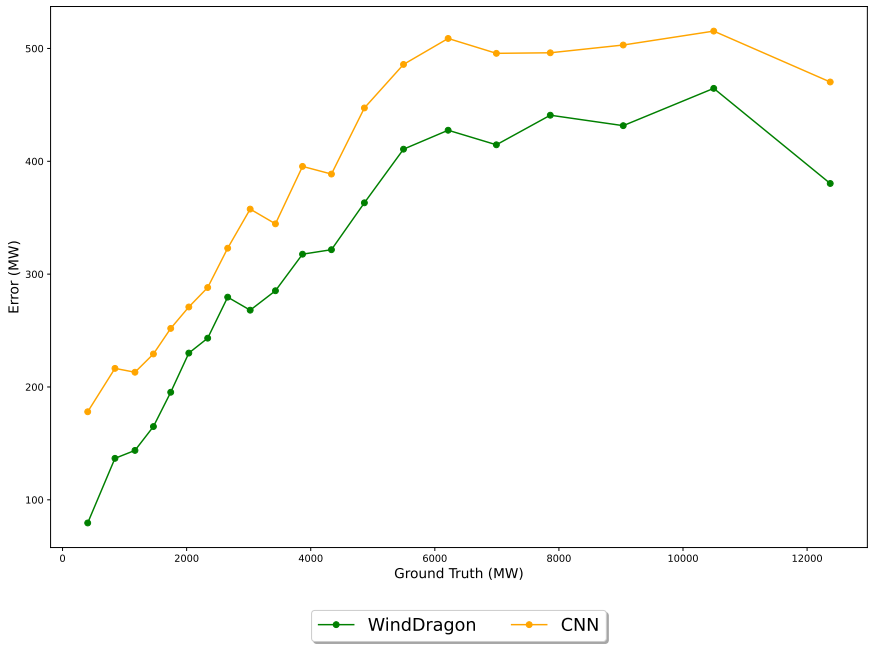
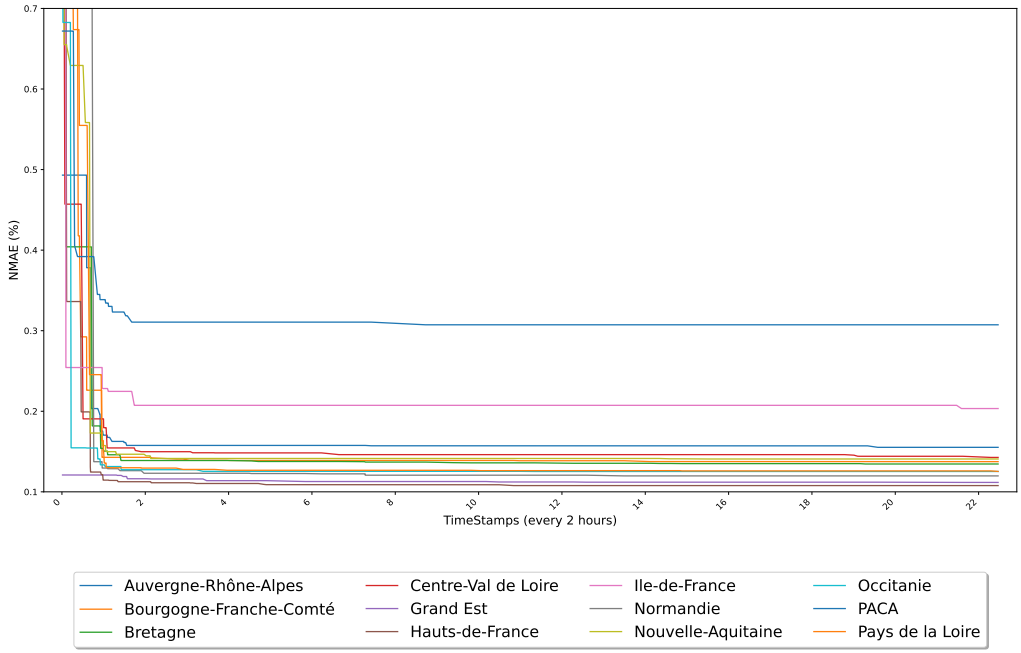
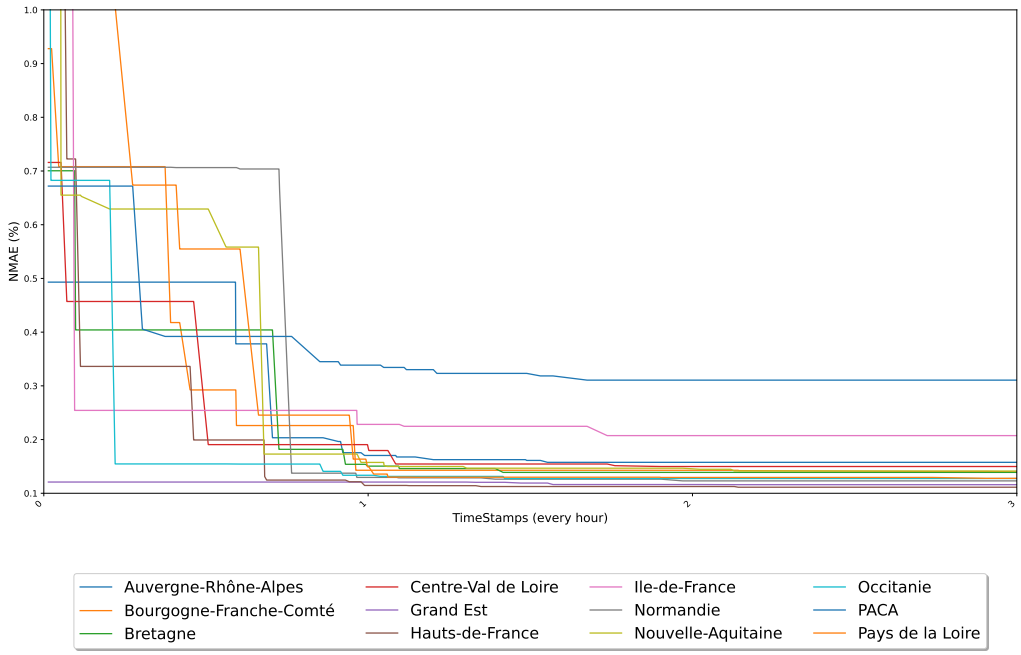


Figure 7.8: Error repartition of the CNN and WindDragon over 20 quantiles.



(a) NMAE through time for each region.



(b) NMAE through time: zoom on the 3 first hours

Figure 7.9: Mutant-UCB convergence: NMAE through time for each region.

7.5 Conclusion

This chapter presents WindDragon, an Automated Deep Learning framework for forecasting regional wind power. WindDragon automated the creation of performing Deep Neural Networks leveraging Numerical Weather Prediction wind speed maps to deliver wind production forecasts. We demonstrate on the french national and regional wind production data that WindDragon can find deep neural networks outperforming traditional and state-of-the-art deep learning models in regional wind power forecasting.

WindDragon, like many AutoML systems is limited by its high running time compared to handcrafted baselines. However, this duration should be compared to the time spent creating powerful models by hand which is often hard to measure. Besides, once the model has been found, the inference speed remains competitive with other deep learning models. Nevertheless, future work could focus on reducing this running training time through even more efficient search algorithms or by reducing the search space. This gained efficiency could also be achieved by reducing the input weather map dimension, through unsupervised representation techniques, for example. The high number of model training and evaluations could be leveraged by creating a mix of models instead of just identifying the best one by region. Section 7.4 highlighted that the baseline models produced quite different forecasts. These differences, if complementary, could enable a mix of models to achieve better performance.

Finally, with the rise of data-driven weather forecasting tools, the accuracy of weather forecasting has increased at various forecast horizons (Ben Bouallègue et al., 2024) and for multiple weather variables. With its non-dependency on past data, our methodology could easily be applied to longer forecast horizons (to be used for other industrial use cases) but also for photovoltaic (PV) regional forecasting, by applying it to solar radiation maps generated by NWP models.

Part IV

Conclusion

Chapter 8

Towards Automated Machine Learning for energy sustainability

8.1 Synthesis

This thesis is a progress towards bridging the gap between AutoML systems and their application to energy sustainability. It focuses on Automated Deep Learning through the development of a Python package called DRAGON, which allows the optimization of neural network architectures and hyperparameters. For each of the identified research directions from Section 1.3.3, here is what was presented in this thesis.

Search Spaces. We have proposed in Chapter 2 a new search space for representing a wide variety of Deep Neural Networks (DNNs). The DRAGON search space can be used to build models of varying complexity using a representation of DNNs in the form of Directed Acyclic Graphs (DAGs). The nodes of these DAGs can be any user-requested *PyTorch* operation whose hyperparameters to optimize are also chosen.

We then applied DRAGON's representation tools to different problems: time series forecasting in Chapter 2, image classification in Chapter 3, load forecasting in regression format in Chapter 5, wind power forecasting using Numerical Weather Predictions (NWP) wind speed maps in Chapter 7, demonstrating the wide variety of search spaces that can be constructed with DRAGON.

The different applications also provided an opportunity to demonstrate the extent to which the search space can be constrained. In Chapter 2, almost no constraints are placed on the search space, which consists of a single DAG for forecasting time series. On the contrary, in Chapter 6, for models combining weather modelling and load forecasting, the search space is much more constrained, with parts of the structure fixed where only the hyperparameters are optimized, and others where it is modelled by a DAG.

DRAGON's multimodal capabilities were also demonstrated, processing tabular data in Chapter 5, images in Chapter 3, weather grids in Chapter 7 and time series in Chap-

ters 2 and 6.

Search Algorithms. The flexibility of the search space, which contains a variety of objects, makes it difficult to optimize. While random search and its variants based on resource allocation such as HyperBand (Li et al., 2018) are directly applicable, classical algorithms such as Bayesian optimization, gradient descent or evolutionary algorithms often rely on notions of proximity between elements, distance, or underlying numerical spaces (potentially high dimensional) where representations could be positioned, which are difficult to define with such objects.

We leveraged the notion of proximity by defining neighborhoods for each type of object in Chapter 2, which can also be seen as mutation operators. For hyperparameters, classical neighborhoods were used (e.g. intervals for integers, other values for categorical variables). Regarding graphs, mutations inspired by the edit distance were developed. These mutations can be applied to our highly flexible structures, allowing nodes or edges to be added or removed, the contents of nodes to be modified, etc.

Thanks to these neighborhoods, it is possible to develop various meta-heuristics such as local search, simulated annealing and evolutionary algorithms. We quickly saw the potential of the latter algorithm on time series, but it seemed rather inefficient for evaluating a large number of configurations because it spent a lot of time evaluating bad solutions. We therefore proposed Mutant-UCB in Chapter 3, an algorithm that uses a UCB strategy to allocate resources, while retaining the mutation of the evolutionary algorithm to intensify the search for good solutions. We have shown that this algorithm is very effective in image classification and wind forecasting, and can be used to evaluate a large number of models.

In Chapter 5 we also explored the possibility of integrating parts of the neural network creation automation into the gradient descent weight optimization, specifically for variable selection. The use case was load forecasting, which relies on a very large number of variables, making the objects very large to optimize using the evolutionary or Mutant-UCB algorithm. Automated deep learning in this case was therefore achieved by mixing an evolutionary approach with an approach similar to that used by DARTS specifically for feature selection. A probability of selection, optimized by gradient descent, was assigned to each of these features.

Creating a tool for electricity forecasters. Most AutoML tools are aimed at a broad audience new to machine learning. They are therefore offered as "no code" packages where the user has to provide very few elements. These approaches are very useful for giving inexperienced people access to state-of-the-art methods. In the context of this thesis, the tool to be developed was aimed at researchers in R&D who have experience using machine learning in developing forecasting tools, but who want to automate decisions that are tedious to make by hand, in order to achieve better performance more quickly. For this reason, as presented in Chapter 4, we have chosen to develop DRAGON

as a generic toolbox that allows the development of specific AutoDL sub-packages for a given application with minimal effort. In Chapters 5, 6 and 7 we showed how to use this toolbox as a domain expert to create "no-code" packages such as EnergyDragon and WindDragon, designed for load and production forecasting respectively.

In the next section, we outline some perspectives for future work aimed at developing DRAGON into an AutoML framework.

8.2 Perspectives: AutoML for energy sustainability

In this thesis, we have developed DRAGON, a tool that initially focuses on the optimization of neural networks. In order to make DRAGON more useful for the forecasting community, but also for others, several research perspectives stand out. First, as presented in Section 1.1.2, many models can be used to forecast production and load. These include GAMs, which are particularly effective for load forecasting and which would benefit from an optimization tool with a representation specific to their particular structure. Secondly, it could be interesting to have search algorithms that allow a choice between different machine learning models (e.g. a search space containing neural networks, GAMs, Random Forests, etc.) for a given problem. Mutant-UCB seems to be a good candidate for this task, as it has the ability to evolve each configuration independently (thus not requiring uniform representations for all configurations) and to evaluate a very large number of models. Chapter 6 also introduced the problem of online learning with correction of the neural network via Kalman filters. With new electricity usages (e.g. sobriety, electrical vehicules) and the growing investment into wind power energy as well as the emergence of new technologies, load and wind power signal distributions and their relation towards explanatory variables change over time. Therefore, high performing models at a given time can become quickly obsolete. Online AutoML, mentioned by Baratchi et al. (2024), is a good avenue of research for this problem. Finally, AutoML takes a lot of computing time and resources. To make DRAGON usable by as many people as possible, and to reduce its energy impact, it is crucial not only to work on its frugality, but also to use it as a global model for several similar forecasting problems, so that only one model needs to be optimized. These ideas are discussed in more detail below.

8.2.1 DRAGAM: optimizing GAMs with DRAGON

GAMs are generally considered to be the state of the art in load forecasting (Antoniadis et al., 2024). In any case, they are widely used by EDF forecasters. In this context, they link the random variable Y to be explained and the explanatory variables X via regular functions h_j (e.g. piecewise polynomial functions, c^2 functions):

$$\mathbb{E}\big[Y \mid X\big] = \sum_j h_j(X)\,.$$

The choice of h_j functions, namely the GAM formula, includes the choice of explanatory variable(s) considered and the regularity. It must be made before the model is trained and

has a major impact on its performance. Typically, the h_j are spline functions of various types (e.g. cubic splines, B-splines, P-splines), where each king can be applied to specific variables, and are parameterized by a complexity level (the spline degree). Few works have been done to fully optimize this structure, while many design choices are left to the user and have a dramatic impact on performance. Chouldechova and Hastie (2015) and Cus (2020) introduced GAMSEL and GAGAM respectively to select the input features and the splines applied to each of them. However, they are very limited in design choices compared to the GAMs developed for load forecasting. Work is in progress on DRAGAM, which will extend DRAGON to provide specific encodings for GAMs. In the same way that DAGs have been used for neural networks, a specific representation will be used for GAMs to automate the choice of explanatory variables, with specific spline types for each type of variable and their level of complexity. The aim is to create a search space able of encompassing all the models used for load forecasting. As with the DAGs, new mutation operators will have to be created so that the DRAGON search algorithms can be reused. The aim of this work is also to automate online learning systems for GAMs, such as the Kalman filter, as has been done for neural networks Chapter 6.

8.2.2 Towards model selection

The extension of DRAGON's tools for creating DRAGAM and optimizing GAMs opens the way to model selection with DRAGON. The idea of allowing a search algorithm to choose between neural networks, GAMs and why not other regression or time series models would be very interesting to explore in the context of forecasting. The user could define search spaces for each type of model, specific to the task at hand, and a search algorithm could select configurations from each of these search algorithms to find the best possible model. Mutant-UCB is an ideal candidate due to its ability to work with complex structures, the fact that it considers each configuration drawn as independent of the others (and therefore does not require a single uniform search space), and its potential to evaluate a large number of models. It would be necessary to reconsider the budget allocated to each configuration (a neural network, for example, takes much longer to train than a GAM). Taking this idea a step further, the objective function could also be revised to search for the best mix of models, rather than the best model. Work such as Gaillard (2015) has shown that mixing successful forecasting models improves the final forecast. A good mix consists of models that provide complementary forecasts. It could be different types of models (for example, a GAM and a random forecast), but also models trained on different types of data (for example, a deep neural network trained on weather maps and a time series model trained solely on past data), or similar models but trained on different training periods with models specialised for forecasting summer or winter data. We could also imagine models trained with different loss functions, including quantile loss functions, squared or absolute errors. An AutoML framework evaluates a very large number of models and could take advantage of these massive

evaluations to find the best mix of structures rather than the best structures. It will then be necessary to find functions that can efficiently identify these best mixes without having to evaluate all possible combinations, which can become very costly.

8.2.3 Online AutoML

Electricity time series, and in particular electricity load, are non-stationary, unstable and may be subject to perturbations (covid, sobriety, new electricity uses, energy crisis). Models therefore need to be constantly adapted (Obst et al., 2021) and their hyper-parameters may also need to be adapted. It can therefore be interesting to approach AutoML in an online-learning context. In this work, we have looked at the adaptation of a neural network with a Kalman filter. This is a first step towards online-learning, but more needs to be done. The optimization of the model and its adaptation is done in a static framework. As mentioned Section 1.2.1, the AutoML task in this thesis consider that the distribution $\mathscr{L}(X, Y)$ is identical between $\mathscr{D}_{\text{train}}$, $\mathscr{D}_{\text{valid}}$ and $\mathscr{D}_{\text{test}}$. However this hypothesis becomes false in the context of these disturbances. The optimal model found in this static framework can therefore quickly become obsolete. Online AutoML has been studied first by Celik et al. (2023), which proposes a flexible and practical AutoML system for online adaptive learning pipelines. Adapting this type of framework to DRAGON could be particularly useful for energy forecasters.

8.2.4 Frugality and explainability

The development of a tool as versatile as DRAGON for the optimization of neural networks was made possible by access to a large number of GPUs. Most of the experiments in this manuscript were carried out on 20 V100 GPUs over several hours. Access to such a large number of GPUs is still quite rare, which may limit the use of DRAGON, and no work has yet been done to reduce the structures obtained so that they are not too cumbersome to use in an operational context. Future work could lead to the gradual pruning of the graphs generated by DRAGON, or at the end of the optimization, in order to limit the structures that are too complex. The aim would be to find the so-called "lottery ticket", introduced by Frankle and Carbin (2018), which hypothesises that there exists a minimal subgraph of a neural network that leads to the same performance. This approach could be applied directly to the graph structure by removing nodes with weights or gradients close to zero, or by penalising the objective function. In addition, simplifying the networks could also make them easier to explain. This idea is very important when using neural networks in a real context. For example, in the context of forecasting for the electricity market, it is important to understand why a model has made a prediction, especially if it is wrong. The importance of variables, for example, can be useful in identifying whether an input variable was inaccurate.

8.2.5 Global models

The idea of frugality can also be approached by creating global models, where a single optimized model is used for different tasks, rather than optimizing one model per task. In Chapter 7, for forecasting wind power, we used the same search strategy to optimize the hyperparameters and architecture for all regions at the same time. This approach avoided the need to run WindDragon on each region. However, we retained the idea of a single model for each time series. Global models go even further in that they apply directly to all tasks. They therefore require consistent inputs and outputs across the different datasets. This kind of approach has been tested in the context of forecasting load at the interface between the transmission and distribution networks (TSO/DSO). There exist more than 2,000 contact points, resulting in 2,000 unique forecasting problems for which it is impossible to have a single model, whether optimized via AutoML or not. The idea was therefore to use a single global model to fit and forecast all the time series simultaneously. We compared several models, including a GAM, a random forest, a simple feed-forward neural network, and a deep learning model optimized with DRAGON. We obtained encouraging preliminary results on a small number of series (about fifty). For each of these 50 series, we were able to build an optimized an unique GAM model and compare it with the global models. The model found by DRAGON was by far the best, outperforming even the local models. We also tested using zero-shot learning to forecast unseen series, an experiment that gave very good results. A paper on this will be published soon.

Bibliography

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.

T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

H. Alkabbani, F. Hourfar, A. Ahmadian, Q. Zhu, A. Almansoori, and A. Elkamel. Machine learning-based time series modelling for large-scale regional wind power forecasting: A case study in ontario, canada. *Cleaner Energy Systems*, 5:100068, 2023.

A. Almalaq and G. Edwards. A review of deep learning methods applied on load forecasting. In *2017 16th IEEE international conference on machine learning and applications (ICMLA)*, pages 511–516. IEEE, 2017.

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.

M. H. Amini, A. Kargarian, and O. Karabasoglu. Arima-based decoupled time series forecasting of electric vehicle charging demand for stochastic power system operation. *Electric Power Systems Research*, 140:378–390, 2016.

A. Antoniadis, J. Cugliari, M. Fasiolo, Y. Goude, and J.-M. Poggi. Statistical learning tools for electricity load forecasting, 2024.

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.

J.-Y. Audibert, S. Bubeck, and R. Munos. Best arm identification in multi-armed bandits. In *COLT*, pages 41–53, 2010.

P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47:235–256, 2002.

N. Awad, N. Mallik, and F. Hutter. Dehb: Evolutionary hyperband for scalable, robust and efficient hyperparameter optimization, 2021.

M. Aziz, J. Anderton, E. Kaufmann, and J. Aslam. Pure exploration in infinitely-armed bandit models with fixed-confidence. In *Algorithmic Learning Theory*, pages 3–24. PMLR, 2018.

A. Ba, M. Sinn, Y. Goude, and P. Pompey. Adaptive learning of smoothing functions: Application to electricity load forecasting. *Advances in neural information processing systems*, 25, 2012a.

A. Ba, M. Sinn, Y. Goude, and P. Pompey. Adaptive learning of smoothing functions: Application to electricity load forecasting. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012b.

Y. Bai, A. Michiorri, and S. Camal. Integrating text analysis in electricity load forecasting: initial findings from uk. In *42nd International Symposium on Forecasting*, 2022.

B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. In *International Conference on Learning Representations*, 2022.

M. Baratchi, C. Wang, S. Limmer, J. N. van Rijn, H. Hoos, T. Bäck, and M. Olhofer. Automated machine learning: past, present and future. *Artificial Intelligence Review*, 57(5):1–88, 2024.

Z. Ben Bouallègue, M. C. Clare, L. Magnusson, E. Gascon, M. Maier-Gerber, M. Janoušek, M. Rodwell, F. Pinault, J. S. Dramsch, S. T. Lang, et al. The rise of data-driven weather forecasting: A first statistical assessment of machine learning–based weather forecasts in an operational-like context. *Bulletin of the American Meteorological Society*, 105(6):E864–E883, 2024.

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.

J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

D. A. Berry, R. W. Chen, A. Zame, D. C. Heath, and L. A. Shepp. Bandit problems with infinitely many arms. *The Annals of Statistics*, 25(5):2103–2116, 1997.

E. Bisong et al. *Building machine learning and deep learning models on Google cloud platform*. Springer, 2019.

S. B. Bosma and N. Nazari. Estimating solar and wind power production using computer vision deep learning techniques on weather maps. *Energy Technology*, 10(8):2200289, 2022.

A. Brock, T. Lim, J. Ritchie, and N. Weston. Smash: One-shot model architecture search through hypernetworks. In *International Conference on Learning Representations*, 2018.

H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang. Efficient architecture search by network transformation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018a.

H. Cai, L. Zhu, and S. Han. Proxylessnas: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2018b.

S. Camal, R. Girard, M. Fortin, A. Touron, and L. Dubus. A conditional and regularized approach for large-scale spatiotemporal wind power forecasting. *Sustainable Energy Technologies and Assessments*, 65:103743, 2024.

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.

E. Campagne, Y. Amara-Ouali, Y. Goude, and A. Kalogeratos. Leveraging graph neural networks to forecast electricity consumption. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases 2024*, 2024.

A. Carpentier and M. Valko. Simple regret for infinitely many armed bandits. In *International Conference on Machine Learning*, pages 1133–1141. PMLR, 2015.

B. Celik, P. Singh, and J. Vanschoren. Online automl: An adaptive automl framework for online learning. *Machine Learning*, 112(6):1897–1921, 2023.

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. *ACM Transactions on Knowledge Discovery from Data*, 19(1):1–23, 2024.

T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

K. Cho, B. van Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, 2014.

A. Chouldechova and T. Hastie. Generalized additive model selection. *arXiv preprint arXiv:1506.03850*, 2015.

J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.

J.-B. Cordonnier, A. Loukas, and M. Jaggi. On the relationship between self-attention and convolutional layers. In *International Conference on Learning Representations*, 2020.

M. Cus. Simultaneous variable selection and structure discovery in generalized additive models. 2020. URL https://github.com/markcus1/gagam/blob/master/GAGAMpaper.pdf. Preprint.

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. G. Carbonell, Q. Le, and R. Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988, 2019.

Z. Dai, G. K. R. Lau, A. Verma, Y. Shu, B. K. H. Low, and P. Jaillet. Quantum bayesian optimization. *Advances in Neural Information Processing Systems*, 36, 2024.

F. Davò, S. Alessandrini, S. Sperati, L. Delle Monache, D. Airoldi, and M. T. Vespucci. Post-processing techniques and principal component analysis for regional wind power and solar irradiance forecasting. *Solar Energy*, 134:327–338, 2016.

J. De Vilmarest and Y. Goude. State-space models for online post-covid electricity load forecasting competition. *IEEE Open Access Journal of Power and Energy*, 9:192–201, 2022.

J. de Vilmarest, J. Browell, M. Fasiolo, Y. Goude, and O. Wintenberger. Adaptive probabilistic forecasting of electricity (net-) load. *IEEE Transactions on Power Systems*, 2023.

A. J. del Real, F. Dorado, and J. Durán. Energy Demand Forecasting Using Deep Learning: Application to the French Grid. preprint, ENGINEERING, Mar. 2020.

D. Deng, F. Karl, F. Hutter, B. Bischl, and M. Lindauer. Efficient automated deep learning for time series forecasting. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 664–680. Springer, 2022.

L. Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, 29(6):141–142, 2012.

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

N. Doumèche, Y. Allioux, Y. Goude, and S. Rubrichi. Human spatial dynamics for electricity demand forecasting: the case of france during the 2022 energy crisis. *arXiv preprint arXiv:2309.16238*, 2023.

G. Dudek. Short-term load forecasting using random forests. In *Intelligent Systems' 2014*, pages 821–828. Springer, 2015.

A. E. Eiben and J. E. Smith. *Introduction to evolutionary computing*. Springer, 2015.

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

N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505*, 2020.

S. Fan and R. J. Hyndman. Short-term load forecasting based on a semi-parametric additive model. *IEEE transactions on power systems*, 27(1):134–141, 2011.

M. Farrokhabadi, J. Browell, Y. Wang, S. Makonin, W. Su, and H. Zareipour. Day-ahead electricity demand forecasting competition: Post-covid paradigm. *IEEE Open Access Journal of Power and Energy*, 9:185–191, 2022.

B. Farsi, M. Amayri, N. Bouguila, and U. Eicker. On Short-Term Load Forecasting Using Machine Learning Techniques and a Novel Parallel Deep LSTM-CNN Approach. *IEEE Access*, 9:31191–31212, 2021. ISSN 2169-3536. doi: 10.1109/ACCESS.2021.3060290.

M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. *Advances in neural information processing systems*, 28, 2015.

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.

T. Firmin, E. Talbi, S. Claudel, and J. Lucas. Hyperparameter optimization of deep neural networks: Application to the forecasting of energy consumption. *PGMO DAYS 2021*, page 45, 2021.

L. Franceschi, P. Frasconi, S. Salzo, R. Grazzi, and M. Pontil. Bilevel programming for hyperparameter optimization and meta-learning. In *International conference on machine learning*, pages 1568–1577. PMLR, 2018.

J. Frankle and M. Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2018.

P. I. Frazier. A Tutorial on Bayesian Optimization. *arXiv:1807.02811 [cs, math, stat]*, July 2018.

P. Gaillard. *Contributions à l'agrégation séquentielle robuste d'experts : travaux sur l'erreur d'approximation et la prévision en loi. Applications à la prévision pour les marchés de l'énergie*. PhD thesis, Université Paris-Sud 11, 2015.

J. Gamot. *Algorithms for Conditional Search Space Optimal Layout Problems*. PhD thesis, Université de Lille, 2023.

E. C. Garrido-Merchán and D. Hernández-Lobato. Dealing with categorical and integer-valued variables in bayesian optimization with gaussian processes. *Neurocomputing*, 380:20–35, 2020.

P. Gijsbers, M. L. Bueno, S. Coors, E. LeDell, S. Poirier, J. Thomas, B. Bischl, and J. Vanschoren. Amlb: an automl benchmark. *Journal of Machine Learning Research*, 25(101):1–65, 2024.

R. W. Godahewa, C. Bergmeir, G. I. Webb, R. Hyndman, and P. Montero-Manso. Monash time series forecasting archive. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

L. Grinsztajn, E. Oyallon, and G. Varoquaux. Why do tree-based models still outperform deep learning on typical tabular data? *Advances in neural information processing systems*, 35:507–520, 2022.

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. Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

W. He. Load Forecasting via Deep Neural Networks. *Procedia Computer Science*, 122: 308–314, 2017. ISSN 18770509. doi: 10.1016/j.procs.2017.11.374.

G. Hebrail and A. Berard. Individual Household Electric Power Consumption. UCI Machine Learning Repository, 2006. DOI: https://doi.org/10.24432/C58K54.

P. Hennig and C. J. Schuler. Entropy search for information-efficient global optimization. *Journal of Machine Learning Research*, 13(6), 2012.

K. Higashiyama, Y. Fujimoto, and Y. Hayashi. Feature extraction of nwp data for wind power forecasting using 3d-convolutional neural networks. *Energy Procedia*, 155:350–358, 2018.

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

M. Hoffman, B. Shahriari, and N. Freitas. On correlation and budget constraints in model-based bandit optimization with application to automatic machine learning. In *Artificial Intelligence and Statistics*, pages 365–374. PMLR, 2014.

M. W. Hoffman and B. Shahriari. Modular mechanisms for bayesian optimization. In *NIPS workshop on Bayesian optimization*, pages 1–5, 2014.

N. Hollmann, S. Müller, K. Eggensperger, and F. Hutter. Tabpfn: A transformer that solves small tabular classification problems in a second. In *The Eleventh International Conference on Learning Representations*, 2023.

T. Hong and S. Fan. Probabilistic electric load forecasting: A tutorial review. *International Journal of Forecasting*, 32(3):914–938, 2016. ISSN 0169-2070. doi: https://doi.org/10.1016/j.ijforecast.2015.11.011. URL https://www.sciencedirect.com/science/article/pii/S0169207015001508.

G. Huang, Y. Li, G. Pleiss, Z. Liu, J. E. Hopcroft, and K. Q. Weinberger. Snapshot ensembles: Train 1, get m for free. In *International Conference on Learning Representations*, 2022.

H. Huang, X. Ma, S. M. Erfani, and J. Bailey. Neural architecture search via combinatorial multi-armed bandit. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5*, pages 507–523. Springer, 2011.

F. Hutter, H. Hoos, and K. Leyton-Brown. An efficient approach for assessing hyperparameter importance. In *International conference on machine learning*, pages 754–762. PMLR, 2014.

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.

International Energy Agency (IEA). Wind power generation, 2023. URL https://www.iea.org/energy-system/renewables/wind. IEA, Paris.

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.

S. M. J. Jalali, G. J. Osório, S. Ahmadian, M. Lotfi, V. M. Campos, M. Shafie-khah, A. Khosravi, and J. P. Catalão. New hybrid deep neural architectural search-based ensemble reinforcement learning strategy for wind power forecasting. *IEEE Transactions on Industry Applications*, 58(1):15–27, 2021.

S. M. J. Jalali, S. Ahmadian, M. Khodayar, A. Khosravi, M. Shafie-khah, S. Nahavandi, and J. P. Catalao. An advanced short-term wind power forecasting framework based on the optimized deep neural network models. *International Journal of Electrical Power & Energy Systems*, 141:108143, 2022.

T. Jasiński. Modeling electricity consumption using nighttime light images and artificial neural networks. *Energy*, 179:831–842, July 2019. ISSN 03605442. doi: 10.1016/j.energy.2019.04.221.

Z. Jian, H. Wenran, Z. Ying, and J. Shufan. Eenas: An efficient evolutionary algorithm for neural architecture search. In *Asian Conference on Machine Learning*, pages 1261–1276. PMLR, 2023.

H. Jin, Q. Song, and X. Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 1946–1956, 2019.

X.-B. Jin, W.-Z. Zheng, J.-L. Kong, X.-Y. Wang, Y.-T. Bai, T.-L. Su, and S. Lin. Deep-Learning Forecasting Method for Electric Power Load via Attention-Based Encoder-Decoder with Bayesian Optimization. *Energies*, 14(6):1596, Mar. 2021. ISSN 1996-1073. doi: $10.3390/\mathrm{en}14061596$.

H. S. Jomaa, J. Grabocka, and L. Schmidt-Thieme. Hyp-rl: Hyperparameter optimization by reinforcement learning. *arXiv preprint arXiv:1906.11527*, 2019.

J. Jonkers, D. N. Avendano, G. Van Wallendael, and S. Van Hoecke. A novel day-ahead regional and probabilistic wind power forecasting framework using deep cnns and conformalized regression forests. *Applied Energy*, 361:122900, 2024.

G. Juberias, R. Yunta, J. G. Moreno, and C. Mendivil. A new arima model for hourly load forecasting. In *1999 IEEE Transmission and Distribution Conference (Cat. No. 99CH36333)*, volume 1, pages 314–319. IEEE, 1999.

K. Kandasamy, G. Dasarathy, J. B. Oliva, J. Schneider, and B. Póczos. Gaussian process bandit optimisation with multi-fidelity evaluations. *Advances in neural information processing systems*, 29, 2016.

K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. P. Xing. Neural architecture search with bayesian optimisation and optimal transport. *Advances in neural information processing systems*, 31, 2018.

G. Kariniotakis. *Renewable energy forecasting: from models to applications*. Woodhead Publishing, 2017.

Z. Karnin, T. Koren, and O. Somekh. Almost optimal exploration in multi-armed bandits. In *International conference on machine learning*, pages 1238–1246. PMLR, 2013.

M. Khodayar and J. Wang. Spatio-temporal graph deep neural network for short-term wind speed forecasting. *IEEE Transactions on Sustainable Energy*, 10(2):670–681, 2018.

M. Khodayar, G. Liu, J. Wang, O. Kaynak, and M. E. Khodayar. Spatiotemporal Behind-the-Meter Load and PV Power Forecasting via Deep Graph Dictionary Learning. *IEEE Trans. Neural Netw. Learning Syst.*, 32(10):4713–4727, Oct. 2021. ISSN 2162-237X, 2162-2388. doi: $10.1109/\mathrm{TNNLS}.2020.3042434$.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017.

R. Kleinberg, A. Niculescu-Mizil, and Y. Sharma. Regret bounds for sleeping experts and bandits. *Machine learning*, 80(2-3):245–272, 2010.

B. Komer, J. Bergstra, and C. Eliasmith. Hyperopt-sklearn: Automatic hyperparameter configuration for scikit-learn. In *Scipy*, pages 32–37, 2014.

N. Koutantos, M. Fotopoulou, and D. Rakopoulos. Automated machine learning for optimized load forecasting and economic impact in the greek wholesale energy market. *Applied Sciences*, 14:9766, 10 2024. doi: $10.3390/\mathrm{app}14219766$.

V. Kovalevsky, N. Zhukova, and A. Tristanov. Building a model of wind turbine power using automl methods. In *International conference Ecosystems without borders*, pages 106–115. Springer, 2022.

O. Kramer and O. Kramer. Scikit-learn. *Machine learning for evolution strategies*, pages 45–53, 2016.

A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. 2009.

R. Lam, A. Sanchez-Gonzalez, M. Willson, P. Wirnsberger, M. Fortunato, F. Alet, S. Ravuri, T. Ewalds, Z. Eaton-Rosen, W. Hu, et al. Learning skillful medium-range global weather forecasting. *Science*, 382(6677):1416–1421, 2023.

H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th international conference on Machine learning*, pages 473–480, 2007.

T. Lattimore and C. Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020.

V. Le Guen. *Deep learning for spatio-temporal forecasting-application to solar energy*. PhD thesis, HESAM Université, 2021.

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, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

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

E. LeDell and S. Poirier. H2o automl: Scalable automatic machine learning. In *Proceedings of the AutoML Workshop at ICML*, volume 2020. ICML San Diego, CA, USA, 2020.

J. Li, K. Cheng, S. Wang, F. Morstatter, R. P. Trevino, J. Tang, and H. Liu. Feature selection: A data perspective. *ACM computing surveys (CSUR)*, 50(6):1–45, 2017.

K. Li, A. Fialho, S. Kwong, and Q. Zhang. Adaptive operator selection with bandits for a multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on Evolutionary Computation*, 18(1):114–130, 2013.

L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670, 2010.

L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018.

N. Li, L. Ma, G. Yu, B. Xue, M. Zhang, and Y. Jin. Survey on evolutionary deep learning: Principles, algorithms, applications, and open issues. *ACM Computing Surveys*, 56 (2):1–34, 2023.

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.

F. Lima, T. I. Ren, and A. Costa. Wind power forecast based on transformers and clustering of wind farms with temporal and spatial interdependence. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 01–06. IEEE, 2022.

M. Lindauer, K. Eggensperger, M. Feurer, A. Biedenkapp, D. Deng, C. Benjamins, T. Ruhkopf, R. Sass, and F. Hutter. Smac3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022.

C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. In *Proceedings of the European conference on computer vision (ECCV)*, pages 19–34, 2018a.

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. In *International Conference on Learning Representations*, 2018b.

H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu. Hierarchical representations for efficient architecture search. In *International Conference on Learning Representations*, 2018c.

H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. In *International Conference on Learning Representations*, 2018d.

X. Liu, J. Zhou, and H. Qian. Short-term wind power forecasting by stacked recurrent neural networks with parametric sine activation function. *Electric Power Systems Research*, 192:107011, 2021a.

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

M. G. Lobo and I. Sanchez. Regional wind power forecasting based on smoothing techniques, with application to the spanish peninsular system. *IEEE Transactions on Power Systems*, 27(4):1990–1997, 2012.

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. L'Heureux, K. Grolinger, and M. A. Capretz. Transformer-based model for electrical load forecasting. *Energies*, 15(14):4993, 2022.

G. Malkomes, C. Schaff, and R. Garnett. Bayesian optimization for automated model selection. *Advances in neural information processing systems*, 29, 2016.

A. A. Mamun, M. Hoq, E. Hossain, and R. Bayindir. A Hybrid Deep Learning Model with Evolutionary Algorithm for Short-Term Load Forecasting. In *2019 8th International Conference on Renewable Energy Research and Applications (ICRERA)*, pages 886–891, Brasov, Romania, Nov. 2019. IEEE. ISBN 978-1-72813-587-8. doi: 10.1109/ICRERA47325.2019.8996550.

M. Massaoudi, I. Chihi, H. Abu-Rub, S. S. Refaat, and F. S. Oueslati. Convergence of Photovoltaic Power Forecasting and Deep Learning: State-of-Art Review. *IEEE Access*, 9:136593–136615, 2021. ISSN 2169-3536. doi: 10.1109/ACCESS.2021.3117004.

A. McNally, C. Lessig, P. Lean, E. Boucher, M. Alexe, E. Pinnington, M. Chantry, S. Lang, C. Burrows, M. Chrust, et al. Data driven weather forecasts trained and initialised directly from observations. *arXiv preprint arXiv:2407.15586*, 2024.

H. Mendoza, A. Klein, M. Feurer, J. T. Springenberg, and F. Hutter. Towards automatically-tuned neural networks. In *Workshop on automatic machine learning*, pages 58–65. PMLR, 2016.

Z. Meng, X. Xie, Y. Xie, and J. Sun. A comparative study of automl approaches for short-term electric load forecasting. In *E3S Web of Conferences*, volume 358, page 02045. EDP Sciences, 2022.

E. S. Miele, N. Ludwig, and A. Corsini. Multi-horizon wind power forecasting using multi-modal spatio-temporal neural networks. *Energies*, 16(8):3522, 2023.

M. Milligan, M. N. Schwartz, and Y.-h. Wan. Statistical wind power forecasting for us wind farms. Technical report, National Renewable Energy Lab., Golden, CO (US), 2003.

M. Mohandes. Support vector machines for short-term electrical load forecasting. *International Journal of Energy Research*, 26(4):335–345, 2002.

G. I. Nagy, G. Barta, S. Kazi, G. Borbély, and G. Simon. Gefcom2014: Probabilistic solar and wind power forecasting using a generalized additive tree ensemble approach. *International Journal of Forecasting*, 32(3):1087–1093, 2016.

R. Nedellec, J. Cugliari, and Y. Goude. Gefcom2012: Electric load forecasting and backcasting with semi-parametric models. *International Journal of forecasting*, 30(2): 375–381, 2014.

Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, A. Y. Ng, et al. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, page 4. Granada, 2011.

A. L. F. Novaes, R. A. d. M. Araujo, J. Figueiredo, and L. A. Pavanelli. A New State-of-the-Art Transformers-Based Load Forecaster on the Smart Grid Domain. *arXiv:2108.02628 [cs]*, Aug. 2021. arXiv: 2108.02628.

J. Nowicka-Zagrajek and R. Weron. Modeling electricity loads in california: Arma models with hyperbolic noise. *Signal Processing*, 82(12):1903–1915, 2002.

D. Obst, J. de Vilmarest, and Y. Goude. Adaptive methods for short-term electricity load forecasting during covid-19 lockdown in france. *IEEE Transactions on Power Systems*, 36(5):4754–4763, 2021. doi: $10.1109/\mathrm{TPWRS}.2021.3067551$.

D. Obst, S. Claudel, J. Cugliari, B. Ghattas, Y. Goude, and G. Oppenheim. Textual data for electricity load forecasting. *Quality and Reliability Engineering International*, 08 2024. doi: $10.1002/\mathrm{qre}.3637$.

R. S. Olson and J. H. Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning*, pages 66–74. PMLR, 2016.

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. In *International Conference on Learning Representations*, 2020.

J. Pathak, S. Subramanian, L. Berkeley, P. Harrington, S. Raja, M. Mardani, T. Kurth, D. Hall, Z. Li, K. Azizzadenesheli, et al. Fourcastnet: A global data-driven high-resolution weather model using adaptive fourier neural operators. *Ann Arbor*, 1001: 48109, 2022.

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.

A. Pierrot and Y. Goude. Short-term electricity load forecasting with generalized additive models. *Proceedings of ISAP power*, 2011, 2011.

P. Pinson, N. Siebert, and G. Kariniotakis. Forecasting of regional wind generation by a dynamic fuzzy-neural networks based upscaling approach. In *EWEC 2003 (European Wind energy and conference)*, pages 5–pages, 2003.

K. N. Pujari, S. S. Miriyala, P. Mittal, and K. Mitra. Better wind forecasting using evolutionary neural architecture search driven green deep learning. *Expert Systems with Applications*, 214:119063, 2023. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2022.119063.

H. Qiu, K. Shi, R. Wang, L. Zhang, X. Liu, and X. Cheng. A novel temporal–spatial graph neural network for wind power forecasting considering blockage effects. *Renewable Energy*, 227:120499, 2024.

A. Rahman, V. Srikumar, and A. D. Smith. Predicting electricity consumption for commercial and residential buildings using deep recurrent neural networks. *Applied Energy*, 212:372–385, Feb. 2018. ISSN 03062619. doi: 10.1016/j.apenergy.2017.12.051.

A. Rawal and R. Miikkulainen. From nodes to networks: Evolving recurrent neural networks, 2018.

E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

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. Schrodi, D. Stoll, B. Ru, R. Sukthanker, T. Brox, and F. Hutter. Construction of hierarchical neural architecture search spaces based on context-free grammars. *Advances in Neural Information Processing Systems*, 36, 2024.

L. Sehovac and K. Grolinger. Deep Learning for Load Forecasting: Sequence to Sequence Recurrent Neural Networks With Attention. *IEEE Access*, 8:36411–36426, 2020. ISSN 2169-3536. doi: 10.1109/ACCESS.2020.2975738.

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.

X. Shang, E. Kaufmann, and M. Valko. A simple dynamic bandit algorithm for hyperparameter tuning. 2019.

O. Shchur, A. C. Turkmen, N. Erickson, H. Shen, A. Shirkov, T. Hu, and B. Wang. Autogluon–timeseries: Automl for probabilistic time series forecasting. In A. Faust, R. Garnett, C. White, F. Hutter, and J. R. Gardner, editors, *Proceedings of the Second International Conference on Automated Machine Learning*, volume 224 of *Proceedings of Machine Learning Research*, pages 9/1–21. PMLR, 12–15 Nov 2023. URL https://proceedings.mlr.press/v224/shchur23a.html.

H. Shi, M. Xu, and R. Li. Deep Learning for Household Load Forecasting—A Novel Pooling Deep RNN. *IEEE Trans. Smart Grid*, 9(5):5271–5280, Sept. 2018. ISSN 1949-3053, 1949-3061. doi: 10.1109/TSG.2017.2686012.

Y. Shu, W. Wang, and S. Cai. Understanding architectures learnt by cell-based neural architecture search. 2020.

N. Siebert. *Development of methods for regional wind power forecasting*. Theses, École Nationale Supérieure des Mines de Paris, Mar. 2008. URL https://pastel.hal.science/tel-00287551.

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

N. Srinivas, A. Krause, S. Kakade, and M. Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. In *Proceedings of the 27th International Conference on Machine Learning*, pages 1015–1022. Omnipress, 2010.

K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Statista Research Department. Europe: Electricity demand per capita 2022. https://www.statista.com/statistics/1262471/per-capita-electricity-consumption-europe/, 2022.

I. Strumberger, E. Tuba, N. Bacanin, R. Jovanovic, and M. Tuba. Convolutional neural network architecture design by the tree growth algorithm framework. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 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.

S. B. Taieb and R. J. Hyndman. A gradient boosting approach to the kaggle load forecasting competition. *International journal of forecasting*, 30(2):382–394, 2014.

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.

Z. Tang, H. Fang, S. Zhou, T. Yang, Z. Zhong, C. Hu, K. Kirchhoff, and G. Karypis. Autogluon-multimodal (automm): Supercharging multimodal automl with foundation models. In K. Eggensperger, R. Garnett, J. Vanschoren, M. Lindauer, and J. R. Gardner, editors, *Proceedings of the Third International Conference on Automated Machine Learning*, volume 256 of *Proceedings of Machine Learning Research*, pages 15/1–35. PMLR, 09–12 Sep 2024. URL https://proceedings.mlr.press/v256/tang24a.html.

J. W. Taylor. An evaluation of methods for very short-term load forecasting using minute-by-minute british data. *International journal of forecasting*, 24(4):645–658, 2008.

K. Terayama, M. Sumita, R. Tamura, and K. Tsuda. Black-box optimization for automated discovery. *Accounts of Chemical Research*, 54(6):1334–1346, 2021.

C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings*

of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 847–855, 2013.

R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 58(1):267–288, 1996.

R. Tu, N. Roberts, V. Prasad, S. Nayak, P. Jain, F. Sala, G. Ramakrishnan, A. Talwalkar, W. Neiswanger, and C. White. Automl for climate change: A call to action. *arXiv preprint arXiv:2210.03324*, 2022.

United Nations Convention on Climate Change. Paris Agreement. Climate Change Conference (COP21), Dec. 2015. URL https://unfccc.int/sites/default/files/english_paris_agreement.pdf.

M. Valko, N. Korda, R. Munos, I. Flaounas, and N. Cristianini. Finite-time analysis of kernelised contextual bandits. In *Uncertainty in Artificial Intelligence*, 2013.

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.

C. Vernade, A. Carpentier, T. Lattimore, G. Zappella, B. Ermis, and M. Brueckner. Linear bandits with stochastic delayed feedback. In *International Conference on Machine Learning*, pages 9712–9721. PMLR, 2020.

J. D. Vilmarest. *State-Space Models for Time Series Forecasting. Application to the Electricity Markets. (Modèles espace-état pour la prévision de séries temporelles. Application aux marchés électriques)*. PhD thesis, Sorbonne University, Paris, France, 2022. URL https://tel.archives-ouvertes.fr/tel-03783480.

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, 2018a.

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, 2019a.

C. Wang, T. Bäck, H. H. Hoos, M. Baratchi, S. Limmer, and M. Olhofer. Automated machine learning for short-term electric load forecasting. In *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 314–321. IEEE, 2019b.

C. Wang, Q. Wu, M. Weimer, and E. Zhu. Flaml: A fast and lightweight automl library. *Proceedings of Machine Learning and Systems*, 3:434–447, 2021.

C. Wang, X. Chen, C. Wu, and H. Wang. Automatic time series forecasting model design based on pruning. *Applied Soft Computing*, page 111804, 2024.

Y. Wang, J.-Y. Audibert, and R. Munos. Algorithms for infinitely many-armed bandits. *Advances in Neural Information Processing Systems*, 21, 2008.

Z. Wang, W. Wang, and B. Wang. Regional wind power forecasting model with nwp grid data optimized. *Frontiers in Energy*, 11:175–183, 2017.

Z. Wang, W. Wang, C. Liu, B. Wang, and S. Feng. Short-term probabilistic forecasting for regional wind power using distance-weighted kernel density estimation. *IET Renewable Power Generation*, 12(15):1725–1732, 2018b.

Z. Wang, W. Wang, C. Liu, and B. Wang. Forecasted scenarios of regional wind farms based on regular vine copulas. *Journal of Modern Power Systems and Clean Energy*, 8(1):77–85, 2019c.

C. White, W. Neiswanger, S. Nolen, and Y. Savani. A study on encodings for neural architecture search. *Advances in neural information processing systems*, 33:20309–20319, 2020.

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.

S. N. Wood. *Generalized additive models: an introduction with R*. CRC press, 2017.

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

J. Yang, G. Jiang, Y. Wang, and Y. Chen. An intelligent end-to-end neural architecture search framework for electricity forecasting model development. *INFORMS Journal on Computing*, 2024.

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.

S. R. Young, D. C. Rose, T. P. Karnowski, S.-H. Lim, and R. M. Patton. Optimizing deep learning hyper-parameters through an evolutionary algorithm. In *Proceedings of the workshop on machine learning in high-performance computing environments*, pages 1–5, 2015.

Y. Yu, X. Han, M. Yang, and J. Yang. Probabilistic prediction of regional wind power based on spatiotemporal quantile regression. In *2019 IEEE industry applications society annual meeting*, pages 1–16. IEEE, 2019.

Y. Yu, M. Yang, X. Han, Y. Zhang, and P. Ye. A regional wind power probabilistic forecast method based on deep quantile regression. *IEEE Transactions on Industry Applications*, 57(5):4420–4427, 2021.

L. Zhang, J. Wen, Y. Li, J. Chen, Y. Ye, Y. Fu, and W. Livingood. A review of machine learning in building load prediction. *Applied Energy*, 285:116452, 2021.

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.

Y. Zhao, C. Stephens, C. Szepesvári, and K.-S. Jun. Revisiting simple regret: Fast rates for returning a good arm. In *International Conference on Machine Learning*, pages 42110–42158. PMLR, 2023.

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.

H. Zhou, S. Zhang, J. Peng, S. Zhang, J. Li, H. Xiong, and W. Zhang. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11106–11115, 2021.

L. Zhou and R. Lu. Attention-based convolutional neural network-long short-term memory network wind power forecasting. In *2023 3rd New Energy and Energy Storage System Control Summit Forum (NEESSC)*, pages 294–297. IEEE, 2023.

L. Zimmer, M. Lindauer, and F. Hutter. Auto-pytorch: Multi-fidelity metalearning for efficient and robust autodl. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(9):3079–3090, 2021.

B. Zoph and Q. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*, 2016.

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.

Appendix A

Appendix of Chapter 2

A.1 Available Operations and Hyperparameters

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

Table A.1: 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: $\text{gelu}(x) = x\mathbb{P}(X \leq x) \approx 0.5x(1 + \tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$
- Softmax: $\sigma(\mathbf{x})_j = \frac{\mathrm{e}^{x_j}}{\sum_{d=1}^{D} \mathrm{e}^{x_d}} \ \forall j \in \{1, \ldots, D\}$

A.2 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 A.2: Information about the Monash datasets (Godahewa et al., 2021).

Appendix B

Appendix of Chapter 3

B.1 Models found by the algorithms

We set the same general seed for each algorithm and data set: CIFAR-10, MRBI and SHVN. This means that the initial pool of solutions is exactly the same for each algorithm and for the three data sets. Hyperband and Random Search found exactly the same architecture, indicating that the best configuration was at the beginning of this pool. Indeed, despite the fact that Hyperband looks at more than twice as many configurations as Random Search, it resulted in the same configuration as Random Search. Hyperband's best results come from the fact that 10 sub-trains are probably too many for this model, which overfits in the case of Random Search. More surprisingly, the best model for Hyperband and Random Search, shown in Figure B.1, is identical for all three data sets.

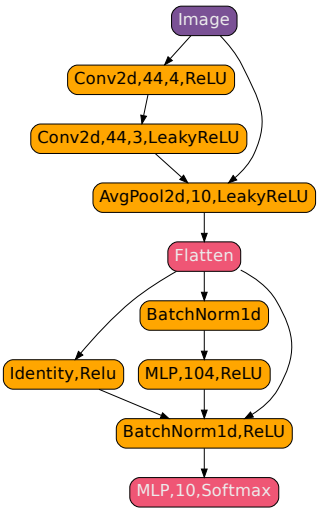
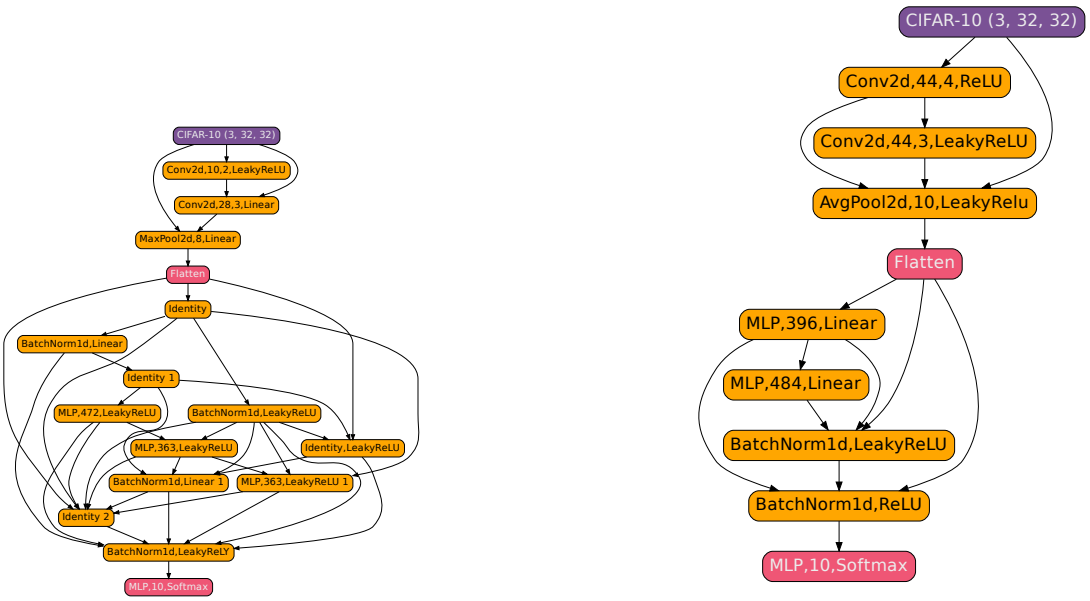


Figure B.1: Best model found for CIFAR-10, MRBI and SVHN by Random Search and Hyperband



(a) Evolutionary algorithm. (b) Mutant-UCB.

Figure B.2: Best configurations found by the evolutionary algorithm and Mutant-UCB on the CIFAR-10 data set.

Thanks to their evolutionary tools, the evolutionary algorithm and Mutant-UCB where able to create more performing configurations. Figure B.2 shows the models found by both algorithms on the CIFAR-10 data set. The one found by Mutant-UCB, in Figure B.2b, is really close to the original one from the pool displayed Figure B.1. The mutation operator was used to add more MLP layers at the end of the neural network which helps improving the model accuracy. On the other hand, the structure shown Figure B.2a, found by the evolutionary algorithm is very different. In this algorithm, a new configuration is created by crossing two parents with the crossover and then applying a mutation to one of the offspring. This double transformation allows to move much further away from the initial pool of solutions. In the case of the CIFAR-10 data set, this led the algorithm to consider very complex structures, which was not necessary to obtain good performance.

B.2 Discussion on the exploration parameter tuning

We discuss here the tuning of the exploration parameter E of Mutant-UCB. We run experiments on the toy MNIST data set (see Deng, 2012) with $T = 10\,000$, $N = 10$ and various values for E. Results are in Table B.1. Figure B.3 shows that the exploration parameter tuning does not impact that much the algorithm performance. Thanks to the limitation in the sub-train allocations, it is difficult to over-exploit, since in this case

Algorithm	Mutant-UCB
$E = 0.01$	2 853 · 98.5
$E = 0.05$	2 828 · 98.7
$E = 1$	2 629 · 97.2
$E = 5$	2 393 · 96.7
$E = 10$	2 435 · 97.6

Table B.1: Number of tested models and accuracy (in %) of the best model for Mutant-UCB run with various exploration parameters.

more and more mutants are created. On the other hand, over-exploration is possible. If E is too large, few models get a lot of sub-trains. While these conclusions shed some light on the parameterization of E, they are not a priori generalizable to any dataset.

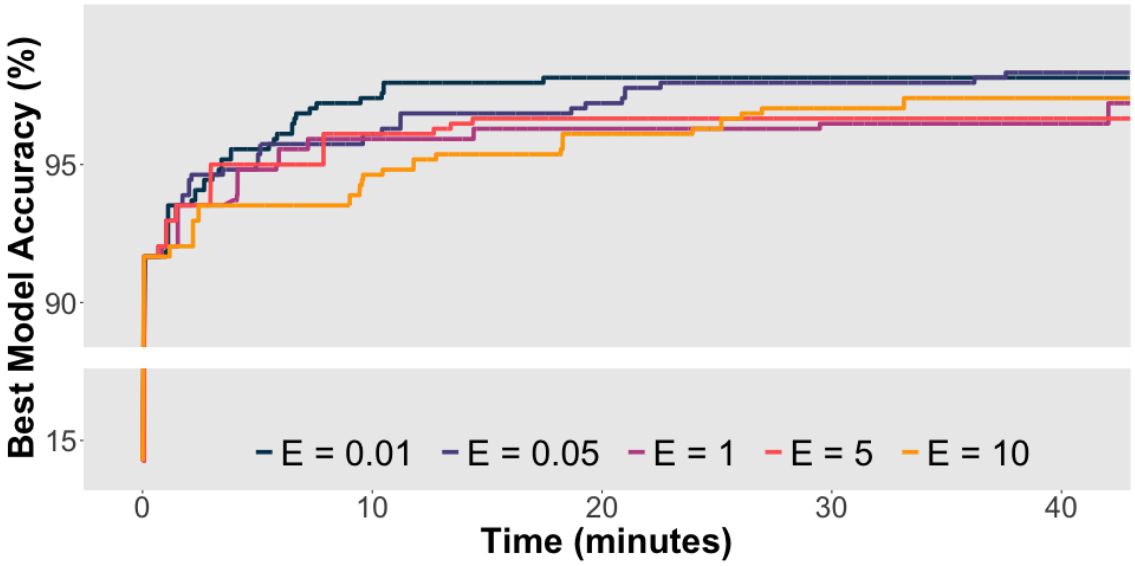


Figure B.3: Maximum accuracy over computation time for the algorithm Mutant-UCB ran with various exploration parameters ($E = 0.01$, $E = 0.05$, $E = 1$, $E = 5$ and $E = 10$) and an initial population of $K = 500$ on MNIST data set.

B.3 Mutants accuracy distributions

We consider here height configurations, trained on the toy MNIST data set (see Deng, 2012), of various accuracies. For each model, we generate and train $4\,000$ mutants. In Figure B.4, we plot in purple the accuracies boxplots of the mutants from this height configurations. These boxplots are ordered according to the accuracy of the parent model. The boxplot in pink represents the accuracies of the random search on MNIST.

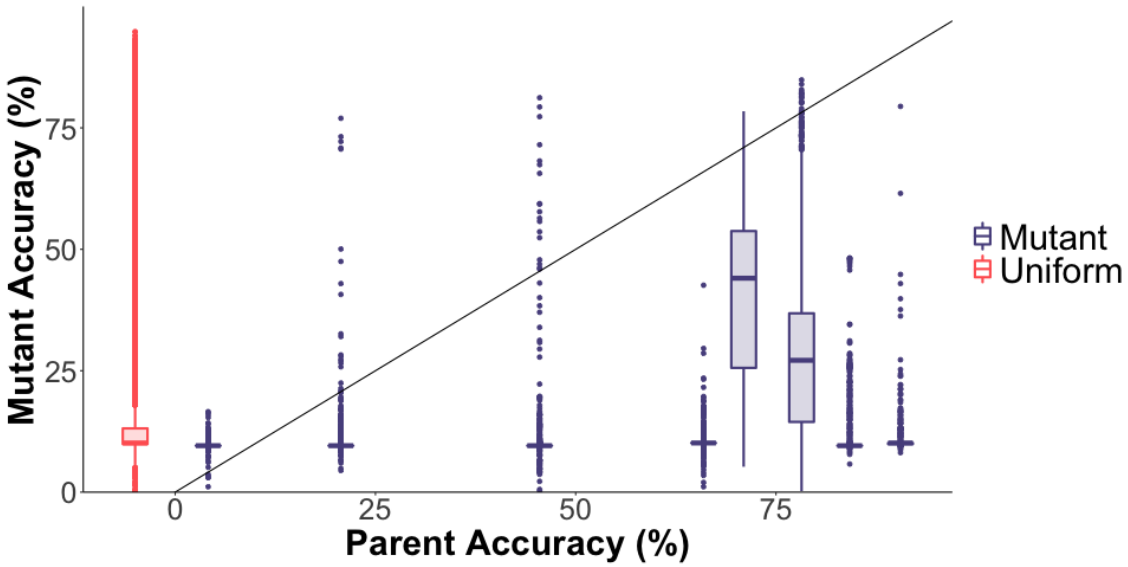


Figure B.4: Mutants distributions for the MNIST data set. For eight configurations with various accuracies we trained and evaluated $4\,000$ mutants. The boxplots in purple represent the accuracies distribution for each configurations, ordered by their accuracy. The boxplot in pink represents the random search.

In black, we plot the function $x \mapsto x$. Empirical means of mutant accuracies do not lie on the first bisector (i.e the black line). This means the equation

$$\mathbb{E}\Big[\, \mu_k \,\Big|\, f_k \,\text{is a mutant of}\, f_j^{\overline{N}_j} \,\Big] = \mu_j \,,$$

does not hold. It is furthermore difficult to come up with any reasonable assumption concerning a potential link between the accuracies of a parent and its mutants. This question requires further study using different data sets and mutation operators. However, this can be costly from a computational perspective.

Appendix C

Appendix of Chapter 5

C.1 Handcrafted DNN for load forecasting

C.1.1 Architecture

Figure C.1 shows the CNN/MLP architecture we have handcrafted for load forecasting. As described Section 5.4, the model has two inputs with different features. One is handled by two parallel convolutions and the other by an MLP. The three branches are then concatenated and fed into another MLP layer. The input data is a bit different than in EnergyDragon, where $X_{\text{EnergyDragon}} \in \mathbb{R}^{H \times F}$. In the CNN/MLP model, the features are split into X_{Conv} and X_{Feed}, which contain F_C and F_F features, respectively, so that $F_C + F_F = F$. Within X_{Conv} and X_{Feed}, the features are concatenated into a one-dimensional vector, i.e. $X_{\text{Conv}} \in \mathbb{R}^{HF_C}$ and $X_{\text{Feed}} \in \mathbb{R}^{HF_F}$.

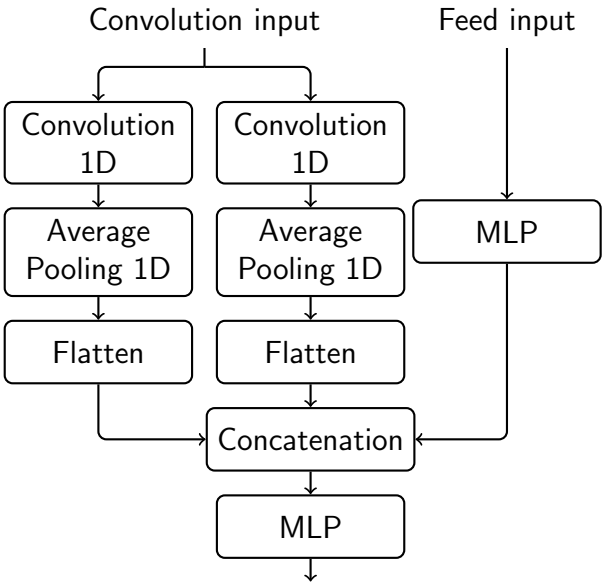


Figure C.1: CNN/MLP Architecture

To represent the CNN/MLP model in EnergyDragon and to include it in the initial population, we had to make some adjustments. The input data is two-dimensional: $X_{\text{EnergyDragon}} \in \mathbb{R}^{H \times F}$, so we set the first DAG Γ_1 to an identity layer and we apply the three branches to the flattened data. The architecture of this new model is shown in Figure C.2. The architecture shown with EnergyDragon gives slightly worse results than the original one.

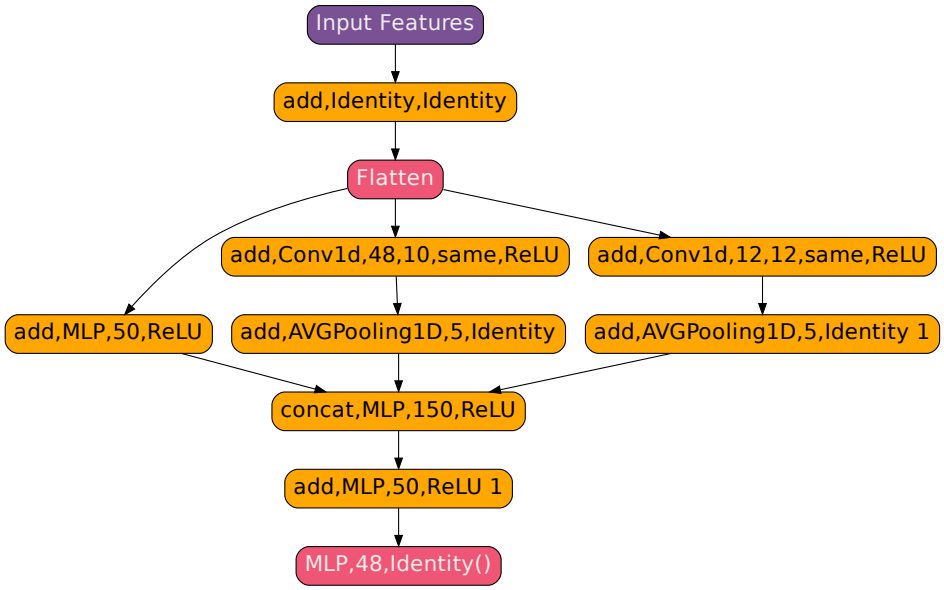


Figure C.2: CNN/MLP Architecture represented with EnergyDragon.

C.1.2 Self-attention

Before creating a fully automated framework for finding neural networks for load forecasting, we searched for DNNs that might be interesting for our problem. The Transformer model (Vaswani et al., 2017), which has recently achieved state-of-the-art results in several areas, naturally caught our attention. We tried to use it on our problem, but without much success: we tried different hyperparameters and the load prediction embedding from the Informer (Zhou et al., 2021), but we could not go below 4% of MAPE. However, one of the major innovations of the Transformer is the self-attention layer. In the vanilla Transformer model, the attention layer is position-invariant, meaning that no assumption is made about the order of the inputs, and the permutation of the input data does not change the result. Therefore, in the original Transformer (Vaswani et al., 2017), the absolute position P_{data} of the data X_{data} in the data set is added to the input

data: $X = X_{\text{data}} + P_{\text{data}}$. The attention scores can then be defined as:

$$A = XW_Q W_K^T X^T = (X_{\text{data}} + P_{\text{data}})W_Q W_K^T (X_{\text{data}} + P_{\text{data}})^T \,,$$

where W_Q and W_K are the query and key weights matrices from the original self-attention.

Later, Dai et al. (2019) introduced relative encoding in their Transformer XL. The idea is to consider only the position difference between query and key instead of the absolute position. They redefined the attention score between a query x_q and a key $x_k \in X_{\text{data}}$:

$$A_{q,k} = x_q^T W_Q^T W_K x_k + x_q^T W_Q^T \hat{W}_K r_{k-q} + u^T W_K x_k + v^T \hat{W}_K r_{k-q} \,,$$

where r_{k-q} is a coding of the relative position, u and v are new attention parameters optimized by backpropagation. From this new attention formulation, Cordonnier et al. (2020) proved that by setting some conditions, the attention layer can be forced to learn as a 2D convolutional layer.

We implemented the attention layers as defined by Cordonnier et al. (2020) for one- and two-dimensional data, and set a parameter "initialization" to indicate whether the layer weights should be initialized to perform a convolution, or if they should be randomized. We replaced the convolutions from the CNN/MLP architecture with these self-attention layers, and compared the performance of three different models: the original CNN/MLP architecture, a self-attention/MLP architecture with the self-attention weights initialized as a convolution, and a self-attention/MLP architecture with randomized self-attention weights. We compared the performance of these models on our data for 10 different seeds. The results are shown in Figure C.3. The models with

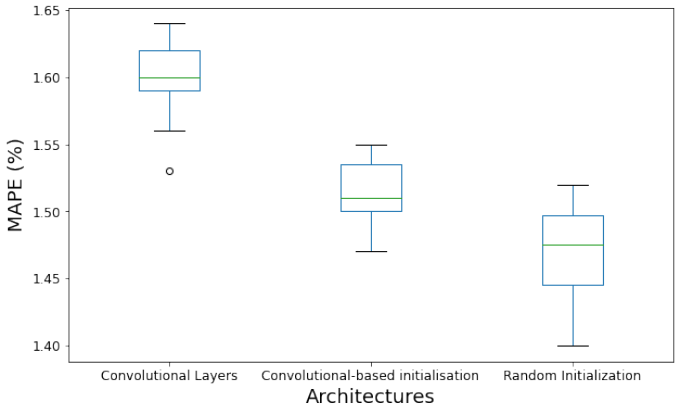


Figure C.3: MAPE on the RTE dataset for three versions of the CNN/MLP model trained with 10 different seeds.

self-attention layers perform better than the original model, reducing the average MAPE from 1.60% for the original model to 1.51% and 1.48% for the convolution and ran-

dom initializations, respectively. This last model even reached a MAPE of 1.40%. We included this self-attention layer in the search spaces of DARTS and EnergyDragon as detailed Appendices C.2 and C.3.

C.2 DARTS

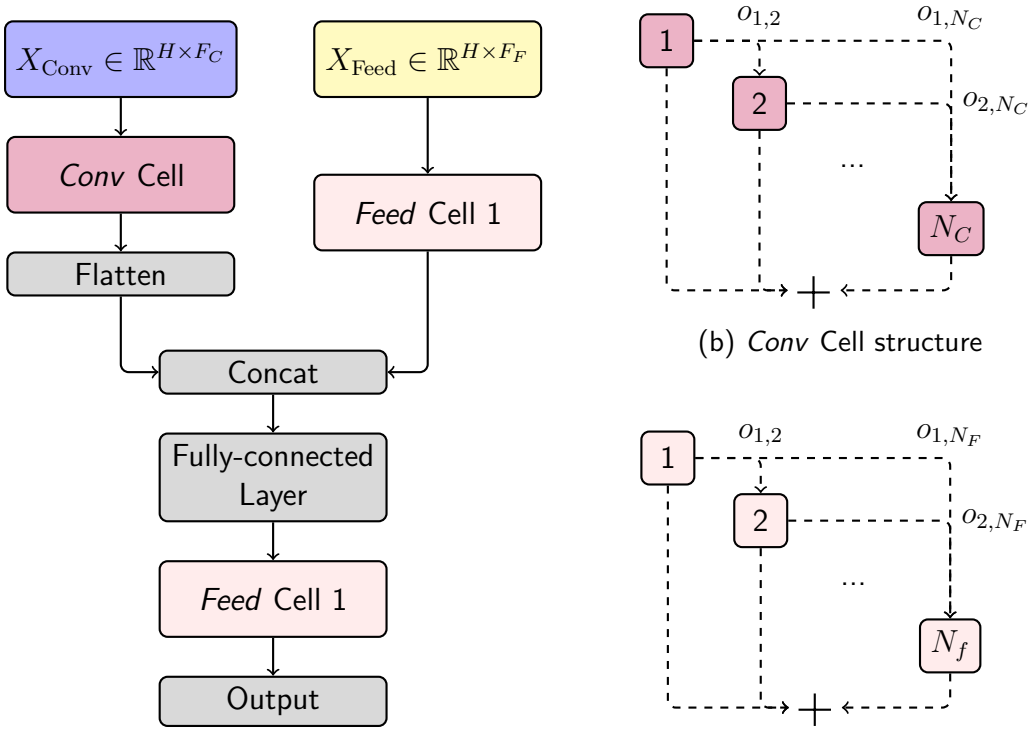
Differential Architecture Search, also called DARTS, was introduced by Liu et al. (2018d), originally for computer vision and NLP tasks. The cell-based search space is composed of either stacked cells to form a convolutional network, or recursively connected cells to form a recurrent network. Each cell is defined as a DAG with N nodes $x^{(i)}$ (latent representations) and edges $O^{(i,j)}$ connecting the nodes (operations). Each DAG has two input nodes and one output. For convolutional cells, the input nodes are the outputs of the two previous cells. For recurrent cells, the input nodes are the input at the current step and the state carried over from the previous step. The cell output is obtained by concatenating all intermediate nodes. An intermediate node is defined as the sum of all previous latent representations after a candidate operation: $x^{(j)} = \sum_{i<j} O^{(i,j)}(x^{(i)})$. The goal of DARTS is to find the best operations between each node. The main idea introduced by DARTS is the relaxation of the search space. A set of candidate operations $\mathcal{O}$ (e.g., convolution, linear, or identity layers) is associated with each connection. Each candidate $o \in \mathcal{O}$ is assigned a probability parametrized by a real $\alpha_o \in \mathbb{R}$ of being part of the final architecture. The relaxed operation between the nodes i and j can be defined as :

$$\bar{o}^{(i,j)} = \sum_{o \in \mathcal{O}} \frac{exp(\alpha_o^{(i,j)})}{\sum_{o' \in \mathcal{O}} exp(\alpha_{o'}^{(i,j)})} o^{(i,j)} \, .$$

At the beginning of the search, the parameters are uniformly initialized for all candidate operations. Then, during model training, the α_o and the network weights are updated alternately using gradient descent. Finally, an argmax function is used to select the candidate operation with the higher probability to build the final architecture. DARTS is a popular framework in the NAS community because it is easy to implement and fast compared to other optimization techniques. It does not require training all new solutions picked from the search space from scratch. The final architecture is a subgraph of the meta-architecture. A drawback of this method is that there is no theoretical guarantee that the optimal subgraph of an optimal meta-architecture is an optimal solution.

Inspired by the work of Chen et al. (2024), where DARTS is used to optimize a DNN for multivariate time series forecasting, we applied DARTS to optimize the CNN/MLP architecture. As in the CNN/MLP model, the general structure of the search space (see Figure C.4a) consists of two inputs. Each input is handled by a dedicated DARTS cell. The *Conv* cell (see Figure C.4b) replaces the two convolutional branches of the

CNN/MLP model. The first *Feed* cell replaces the first MLP branch and the Second replaces the last MLP branch. Each cell has a maximum of $N_c = 4$ nodes and the candidate operations $o_{i,j}$ on each connection are those from the original architecture: e.g., Conv1d/Pooling layers for the Conv cell (see Figure C.4b) or MLP/Identity layers for the Feed cells (see Figure C.4c). We also added zero operations and Attention layers as defined Appendix C.1.2 to our search space, with the same hyperparameters as the convolutional layers.



(a) General Structure

(b) *Conv* Cell structure

(c) *Feed* Cell Structure

Figure C.4: DARTS Search Space for load forecasting.

C.3 EnergyDragon search space details

The layers (operations) used in our search space are detailed Table C.1. Most layers are adapted or used in both Γ_1 (two-dimensional data) and Γ_2 (one-dimensional data), except for the Temporal Attention and the Spatial Attention, which are specific to Γ_1. Given an input data $X \in \mathbb{R}^{H \times f \times d}$, where f and d would be two latent dimensions within the DNN, the attention matrix is the same as defined Appendix C.1.2:

$$\text{Attention}(X) = \text{softmax}(XW_QW_K^TX^T + XW_Q\hat{W}_K^T\delta_R + uW_K^TX^T + v\hat{W}_k^T\delta_R)XW_O + b_O\,,$$

Layer type	Optimized hyperparameters	
Identity	-	
Fully-Connected (MLP)	Output shape	Integer
Self-Attention	Operation dimension	[temporal, spatial] (Γ_1 only)
	Initialization type	[convolution, random]
	Heads number	Integer
	Output dimension	Integer
1D/2D Convolution	Kernel size	Integer
	Output dimension	Integer
1D/2D Pooling	Pooling size	Integer
	Pooling type	[Max, Average]
1D/2D Normalization	Normalization type	[Batch, Layer]
Dropout	Dropout rate	Float

Table C.1: Layers available and their associated hyperparameters in the EnergyDragon search space (for Γ_1 and Γ_2). The self-attention layer is explained Appendix C.1.2)

with W_Q and W_K the query and key weight matrices. In the Temporal Attention case, $W_Q, W_K \in \mathbb{R}^{H \times d \times N_h \times 2}$ and in the Spatial attention case, $W_Q, W_K \in \mathbb{R}^{f \times d \times N_h \times 2}$, where $N_h \in \mathbb{N}^+$ is the head number. The Temporal Attention computes attention scores between the time steps as depicted Figure C.5a, whereas the Spatial Attention computes attention scores between the features, as shown Figure C.5b.

C.4 Additional experimental results

In the Section we give more information one the experimental results presented Section 5.4. The Appendix C.4.1 detail the different architectures and hyperparameters found by the versions of EnergyDragon presented Section 5.4. Appendix C.4.2 present the forecast of various algorithms from our baseline over the last week of November. The EnergyDragon convergences over time are given Figure C.12. Finally, Appendix C.4.4 discuss the features use by the GAM, the CNN/MLP and the different versions of EnergyDragon.

C.4.1 Models found by the algorithms

Respectively Figures C.6, C.7, C.8, C.9 and C.10 represent the architectures and hyperparameters found by ED RS, ED SSEA, ED SSEA Crossover, ED SSEA Crossover CN-

$x_{1,4}$	$x_{2,4}$	$x_{3,4}$	$x_{4,4}$	$x_{5,4}$	$x_{6,4}$
$x_{1,3}$	$x_{2,3}$	$x_{3,3}$	$x_{4,3}$	$x_{5,3}$	$x_{6,3}$
$x_{1,2}$	$x_{2,2}$	$x_{3,2}$	$x_{4,2}$	$x_{5,2}$	$x_{6,2}$
$x_{1,1}$	$x_{2,1}$	$x_{3,1}$	$x_{4,1}$	$x_{5,1}$	$x_{6,1}$

(a) Temporal attention: the attention scores are computed along the time axis. There is no inter-variable interactions

$x_{1,4}$	$x_{2,4}$	$x_{3,4}$	$x_{4,4}$	$x_{5,4}$	$x_{6,4}$
$x_{1,3}$	$x_{2,3}$	$x_{3,3}$	$x_{4,3}$	$x_{5,3}$	$x_{6,3}$
$x_{1,2}$	$x_{2,2}$	$x_{3,2}$	$x_{4,2}$	$x_{5,2}$	$x_{6,2}$
$x_{1,1}$	$x_{2,1}$	$x_{3,1}$	$x_{4,1}$	$x_{5,1}$	$x_{6,1}$

(b) Spatial attention: the attention scores are computed along the feature axis. There is no intra-variable interactions

Figure C.5: Spatial and Temporal Attentions vizualisation, applied on $X = \{x_t\}_{t=1}^{H} = \{x_i\}_{i=1}^{F} \in \mathbb{R}^{H \times F}$.

N/MLP and the best algorithm ED SSEA CNN/MLP. The model found by the Random Search is very simple and based only on the Spatial Attention layer defined Appendix C.3, which computes attention scores between the features. All the found architectures use

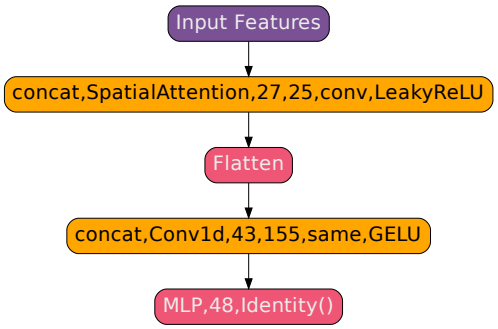


Figure C.6: Architecture found by ED RS. Best MAPE=1.374%.

the attention layers presented Appendix C.1.2. This confirms our first experiments showing that our self-attention layers is efficient for load forecasting. The architectures found by the versions with crossover are more complex, with more layers and connections than the versions without crossover. The appearance of many identity layers invites us to think about pruning our graphs for future work on EnergyDragon. It seems that certain nodes are not necessarily useful, and it might be a good idea to automatically remove them during optimization to avoid ending up with architectures that are too complex or difficult to interpret. The best architecture, which achieves a MAPE of 1.131% as shown in Figure C.10, is quite simple. Finally, it is difficult to conclude whether including the CNN/MLP as input to EnergyDragon affects the architecture found. Our intuition is that including the CNN/MLP probably has an influence on the selected features. A deeper discussion can be found Appendix C.4.4.

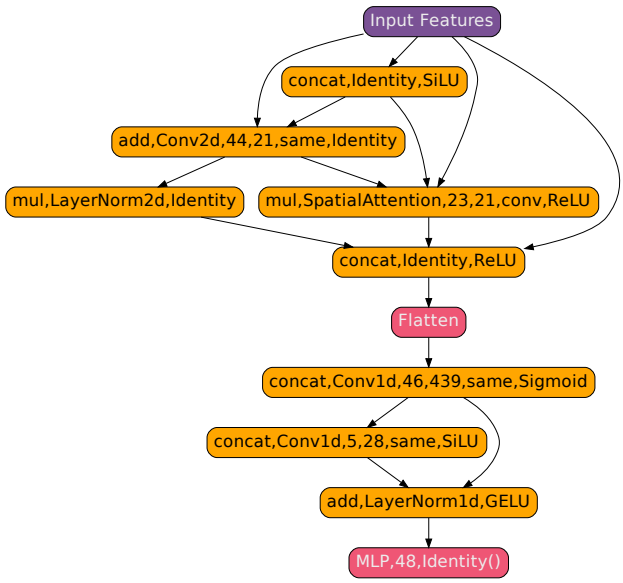


Figure C.7: Architecture found by ED SSEA. Best MAPE=1.258%.

C.4.2 Weekly comparative visuals of all baseline forecasts

Forecasts from various models of our baseline for the last week of November can be found in Figure C.11. Most models have the same shape and, like GAM, overpredict during this week. AutoPytorch without the traditional baseline produces a constant signal as shown in Figure C.11d, which explains its poor MAPE and RMSE. We compare in Figure C.11b the forecast of DARTS with the output of the CNN/MLP model. Using DARTS allowed to improve the overprediction of CNN/MLP, but DARTS is still higher than ED SSEA CNN/MLP as shown in Figure C.11c.

C.4.3 EnergyDragon convergence

Figure C.12 shows the loss of the best model over time for ED RS, ED SSEA, ED SSEA Crossover, ED SSEA Crossover CNN/MLP, and ED SSEA CNN/MLP. We can see that most versions converge to their best model in less than 10 hours, even if we let the algorithm run for another 10 hours. The versions that include the CNN/MLP in their initial population converge much faster than the version without. ED SSEA CNN/MLP, which gave the best result, converged in a little more than 4 hours.

C.4.4 Features

Part of the features used in our experiments cannot be revealed due to industrial confidentiality. Therefore, we have renamed our 33 features from f_0 to f_33. Some are weather variables like temperature or wind. Others are general calendar features: the

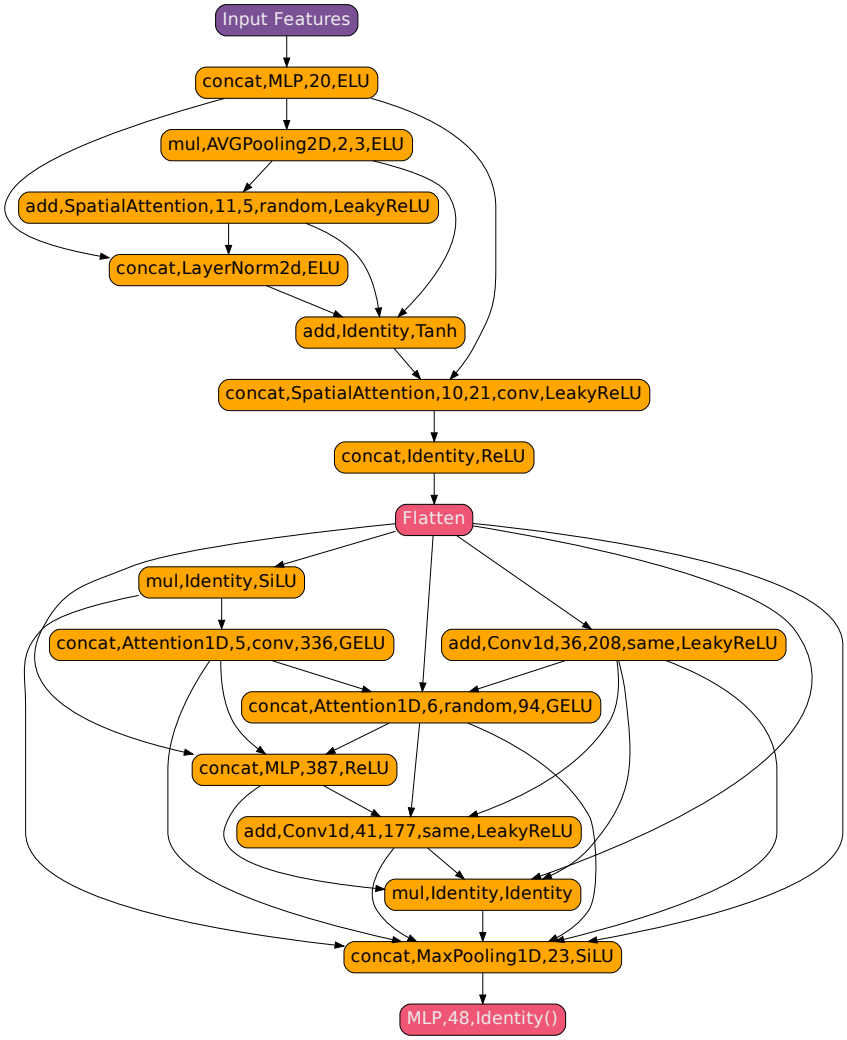


Figure C.8: Architecture found by ED SSEA Crossover. Best MAPE=1.190%.

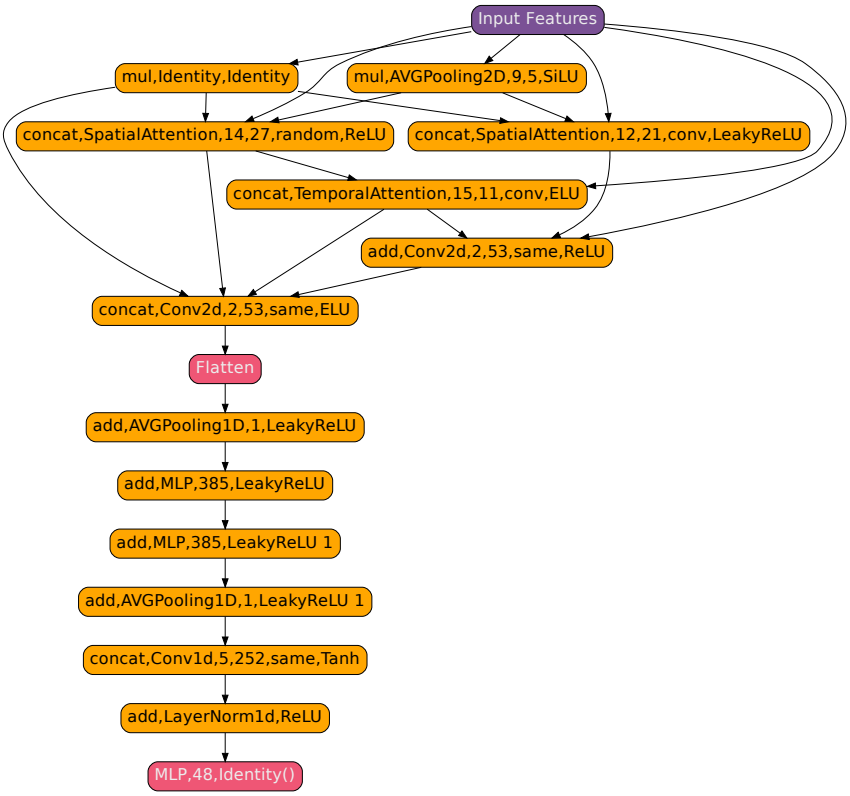


Figure C.9: Architecture found by ED SSEA Crossover CNN/MLP. Best MAPE=1.182%.

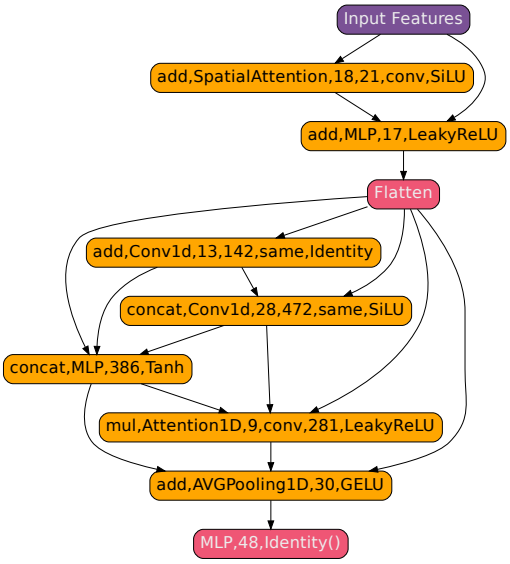
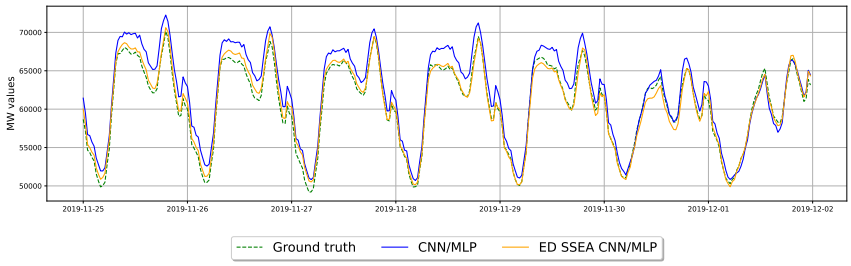
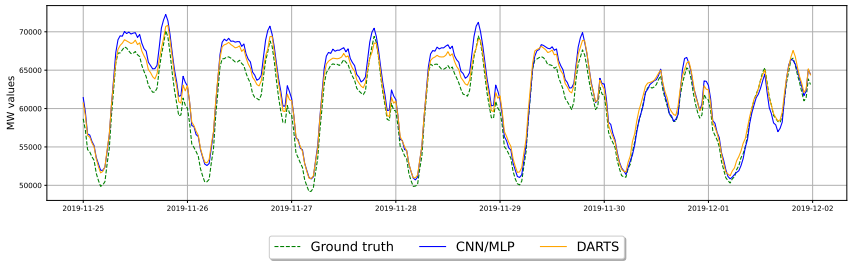


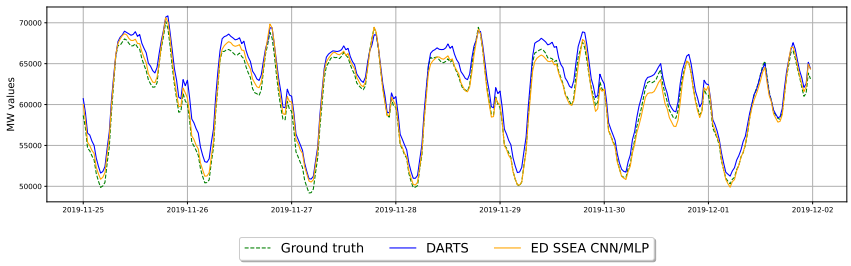
Figure C.10: Architecture found by ED SSEA CNN/MLP. Best MAPE=1.131%



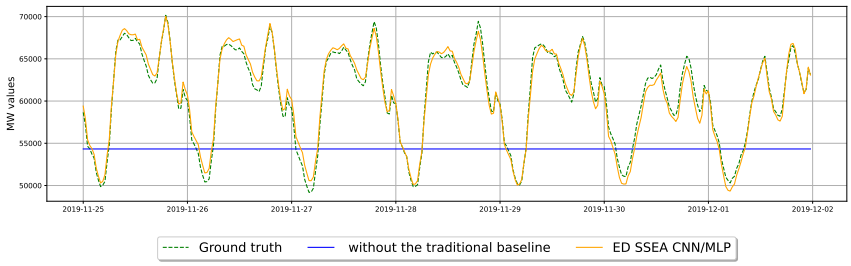
(a) CNN/MLP vs ED SSEA CNN/MLP



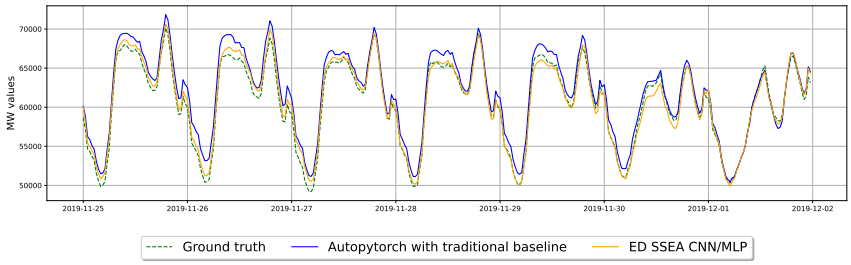
(b) DARTS vs CNN/MLP



(c) DARTS vs ED SSEA CNN/MLP



(d) AutoPytroch vs ED SSEA CNN/MLP



(e) AutoPytroch with the traditional baseline vs ED SSEA CNN/MLP

Figure C.11: Comparison of the forecasts from various algorithm over the last week of November 2019.

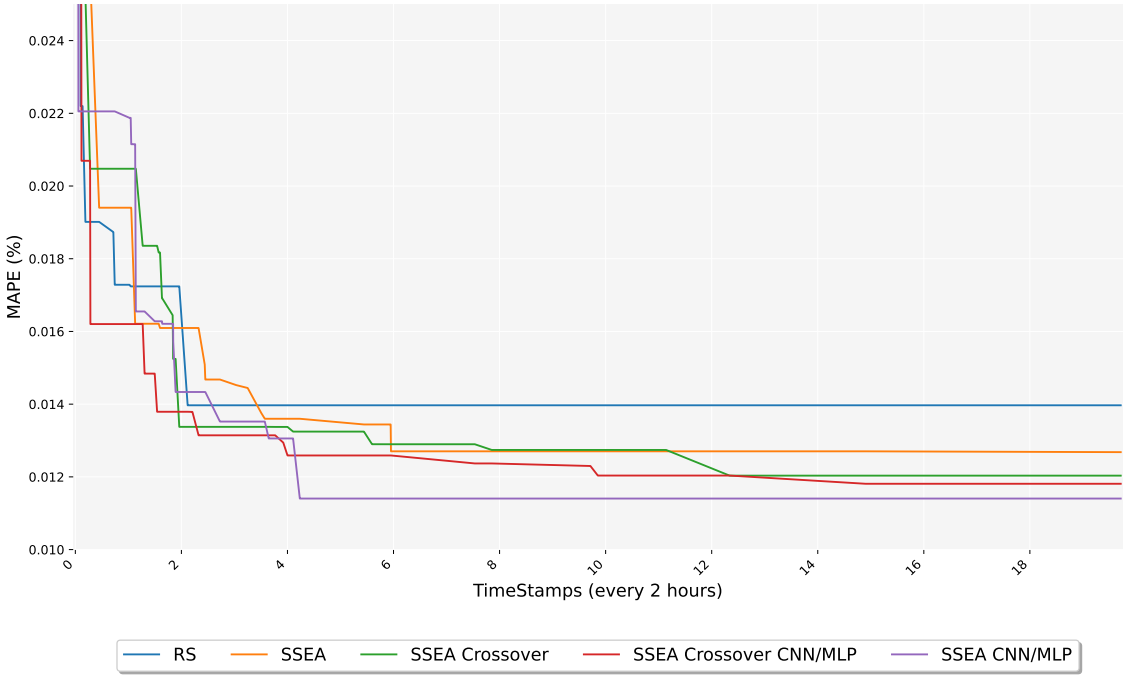


Figure C.12: Loss of the best model over time for the different versions of EnergyDragon used in our experiments Section 5.4.

month, the week of the day, or more related to France, like the holidays or the time shift. We present in Figure C.13 the features selected by the models. The GAM use very different features compared to the other models, as stated Section 5.1. All versions of EnergyDragon selected a number of features close to the number of features used by GAM and CNN/MLP. This means that our LASSO penalization is efficient. Finally, the versions of EnergyDragon with CNN/MLP as input selected a number of features close to the number used by CNN/MLP. As mentioned Appendix C.4.1, this can explain the faster convergence of the version with CNN/MLP as input.

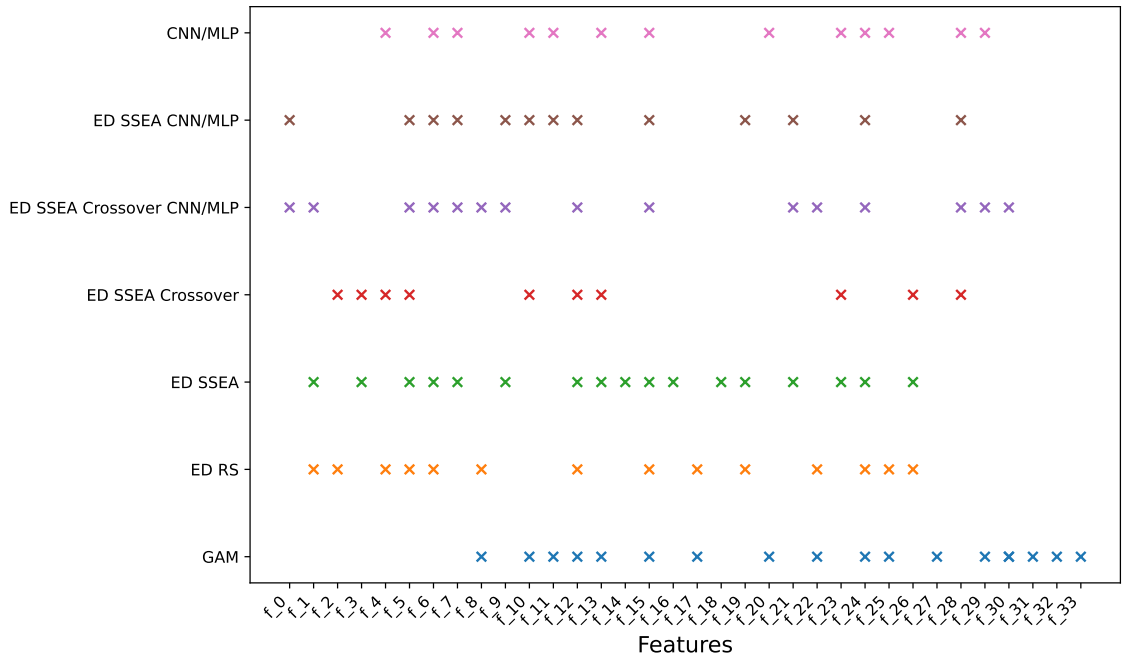


Figure C.13: Features selected by the GAM, CNN/MLP and the various versions of EnergyDragon used in our experiments. The features name cannot be revealed due to the industrial confidentiality, and are renamed to $f_0, \ldots, f_{33}$.

C.5 Norwegian Use Case

In this Section, we present the results of a reduced baseline on another case study: the hourly Norwegian Load for the year 2018.

C.5.1 Dataset

The dataset comes from the ENTSO-E (European Network of Transmission System Operators for Electricity) transparency platform[1] and contains the Norwegian national load data at hourly intervals. Each day contains $H = 24$ time steps. We trained our models from 2014 to 2017 and compare the performance to the year 2018 using the MAPE (ℓ_{MAPE}). For this dataset, we have access to 34 variables, including weather and calendar features. We have also anonymized these features in the following plots.

C.5.2 Baseline

We reduced the full baseline presented Section 5.4 by taking AutoPytorch with the traditional baseline (AutoPytorch), an open-source regression model, the Generalized Additive Model (GAM), and EnergyDragon with the steady-state algorithm, with and

[1] https://transparency.entsoe.eu/load-domain/r2/totalLoadR2/show?

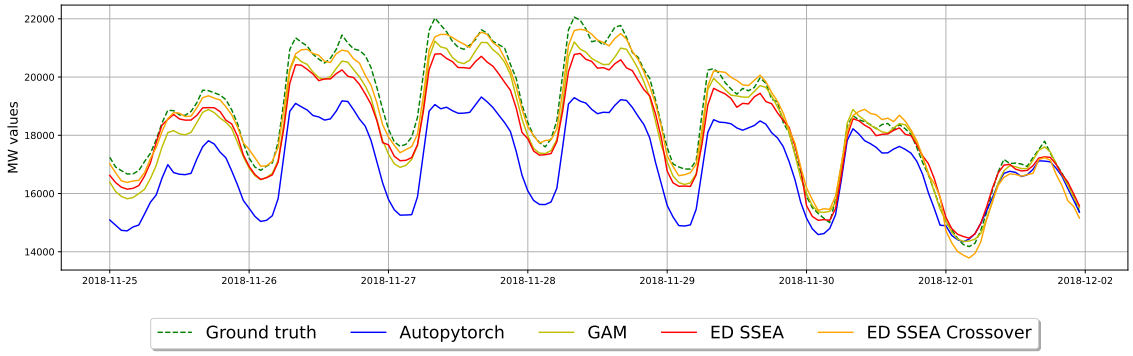


Figure C.14: Norwegian load power forecasting for the last week of November 2018. The ground truth is displayed in dotted line.

without the crossover (ED SSEA and ED SSEA Crossover). The setup for each model from the baseline is the same as Section 5.4.

Model	MAPE	RMSE (in MW)
GAM	2.430%	474.0
AutoPytorch	3.429%	660.7
ED SSEA	2.196%	465.7
ED SSEA Crossover	**2.019%**	**426.3**

Table C.2: MAPE and RMSE of the different models from our baseline on theNorwegian dataset. The reference model is the GAM and the best model is highlighted in bold.

C.5.3 Results

The results can be found in Table C.2. They are similar to those found for the French use case. AutoPytorch with the traditional baseline is the worst model, with a MAPE 41% higher than the reference MAPE given by the GAM model. EnergyDragon was able to improve the GAM forecast even without the inputs from the CNN/MLP model. As in the French case, the crossover helped the algorithm to converge to a better result. This last model produced the best result, improving the GAM MAPE by 17%. The RMSE indicates that Norwegian residents consume less electricity than French residents. The comparison of the forecast for all models can be found Figure C.14. All models got the correct curve shape, but except for the ED SSEA Crossover model, all models underestimated the load.

C.5.4 EnergyDragon Results Analysis

Figure C.15 and Figure C.16 show the DNNs found by ED SSEA and ED SSEA Crossover respectively on the Norwegian dataset. The conclusions of Appendix C.4.1 do not hold anymore, since the best model found by ED SSEA Crossover does not use an attention layer. This dataset is a bit simpler, with only 24 values per day, resulting in a smoother signal than the French load with 48 values per day, and it can explain the use of Convolution layers instead of Attention layers to produce smoother predictions. The two DNNs are very different, with no common substructures, but compared to the french use case, the version with crossover did not produced an overly complicated DNN compared to the version without crossover. The features selected by both versions of EnergyDragon

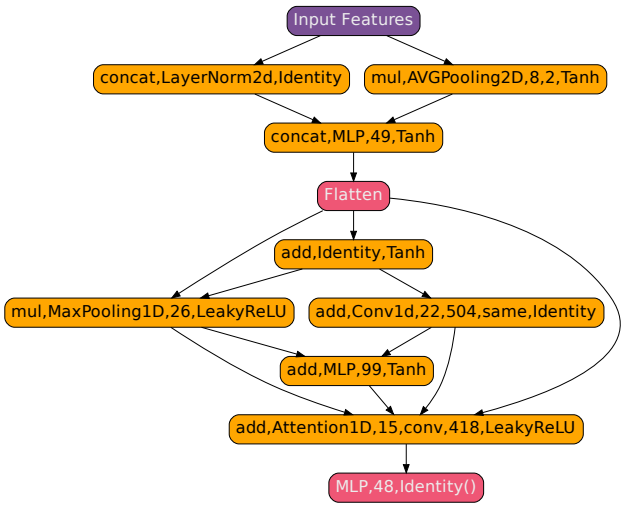


Figure C.15: Architecture found by ED SSEA on the Norwegian dataset.

can be found Figure C.17. The version without crossover selected 17 features, whereas the version with the crossover selected 13 features. Both models only have 6 features in common. Finally, Figure C.18 shows the loss of the best model found so far over time for ED SSEA and ED SSEA Crossover. Both models converged in less than 5 hours, even if we let them run for another 20 hours. The crossover version converged very quickly, in less than an hour and a half. This fast convergence can explain the simple DNN compared to the DNNs obtained with the Crossover versions for the French use case. This good DNN was found before the algorithm used too many successive crossovers.

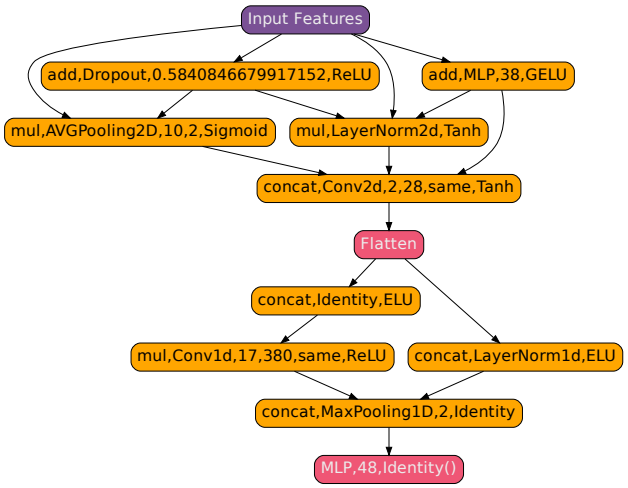


Figure C.16: Architecture found by ED SSEA Crossover on the Norwegian dataset.

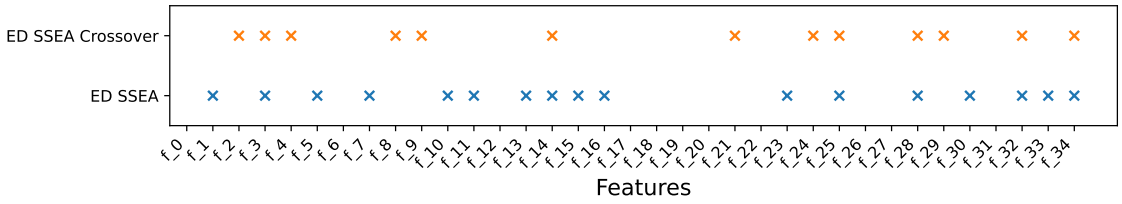


Figure C.17: Features selected by ED SSEA and ED SSEA Crossover for the Norwegian dataset. The features name cannot be revealed due to the industrial confidentiality, and are renamed to. $f_0, \ldots, f_{34}$.

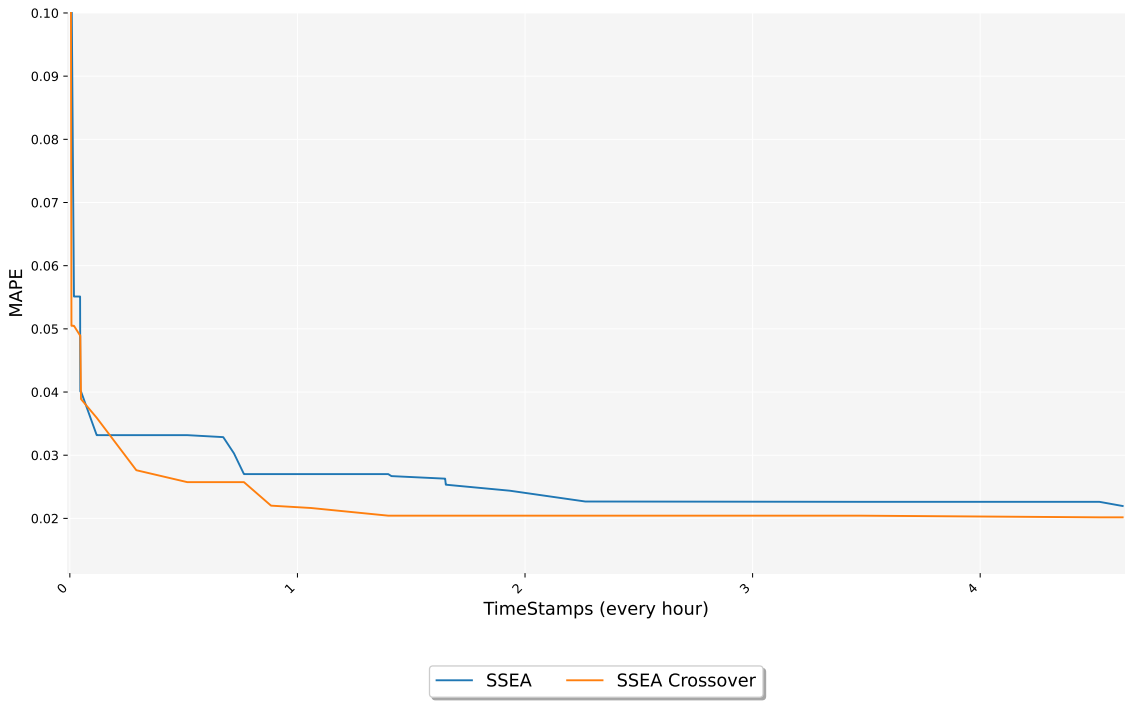


Figure C.18: Loss of the best model over time for the different versions of EnergyDragon on the Norwegian dataset.

Appendix D

Appendix of Chapter 7

D.1 Models found by WindDragon for various regions

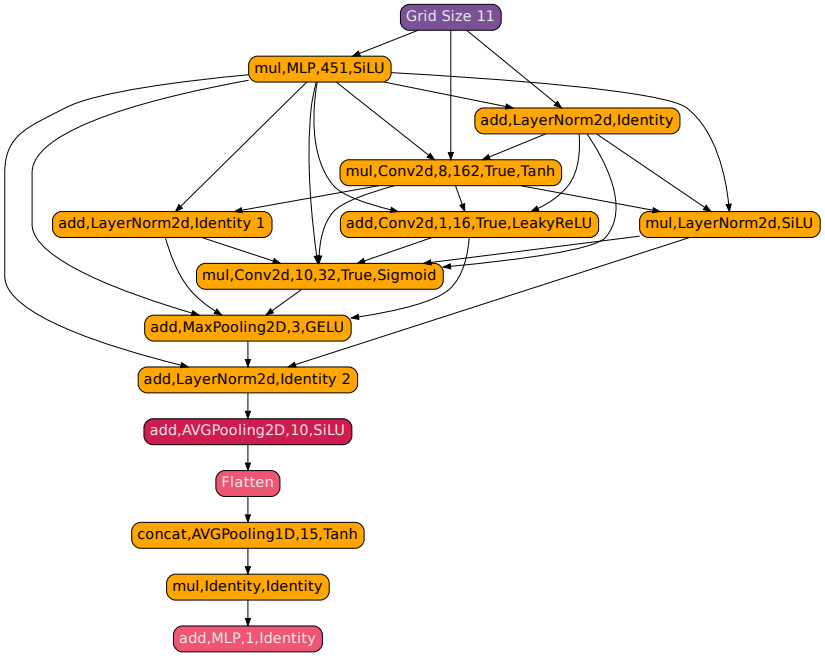


Figure D.1: Architecture found by WindDragon on Grand Est.

D.2 Forecasts comparison

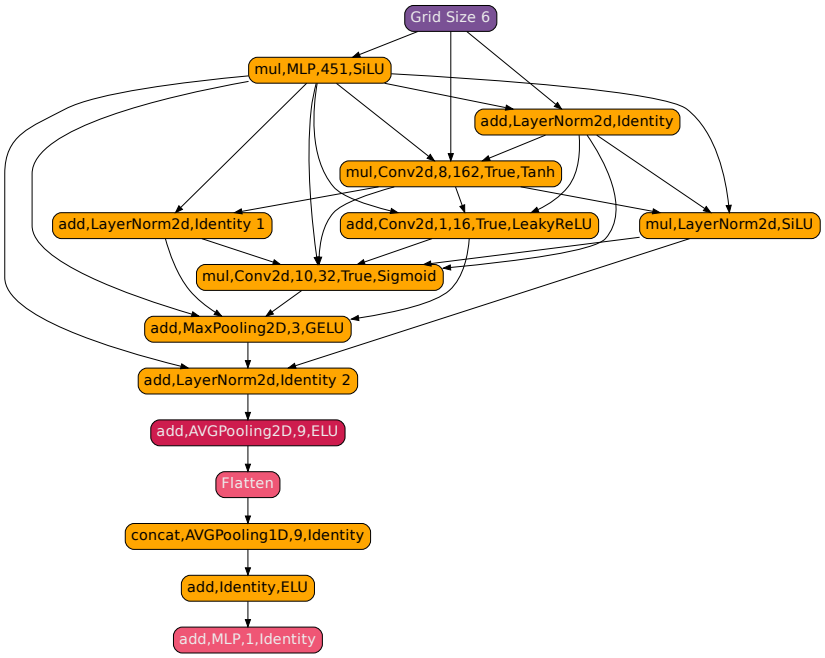


Figure D.2: Architecture found by WindDragon on Auvergne-Rhône-Alpes.

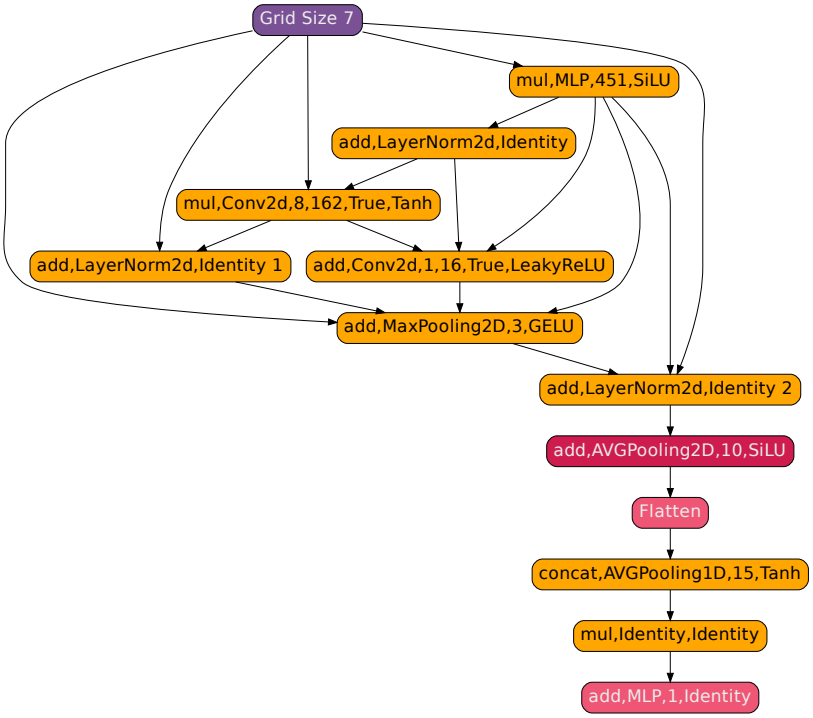


Figure D.3: Architecture found by WindDragon on Hauts-de-France.

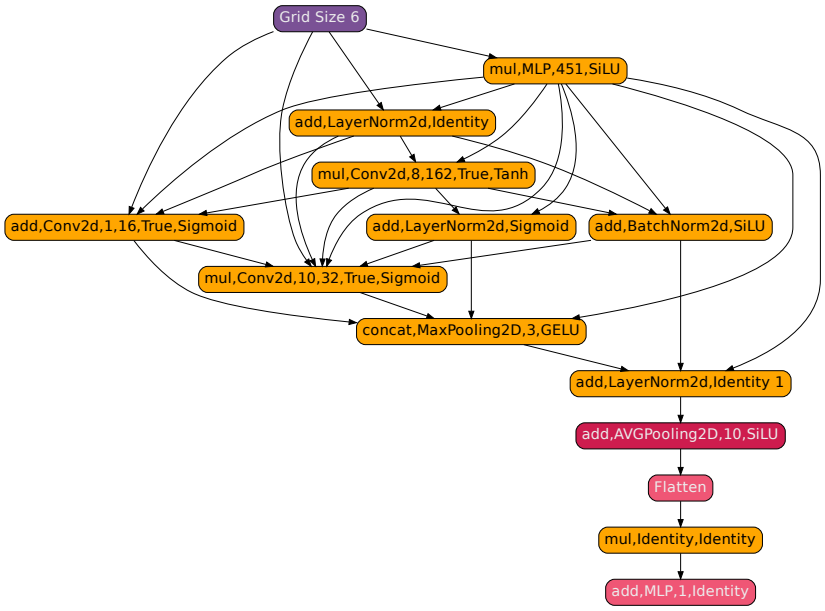


Figure D.4: Architecture found by WindDragon on Île-de-France.

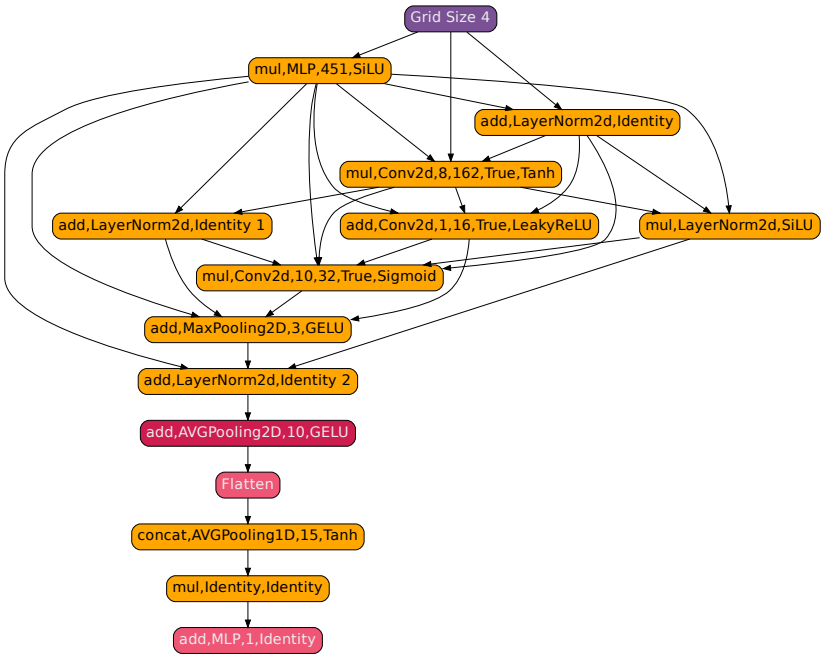
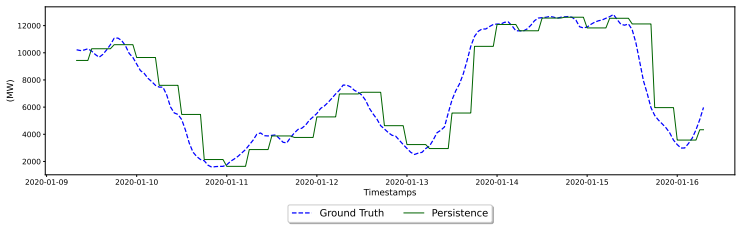
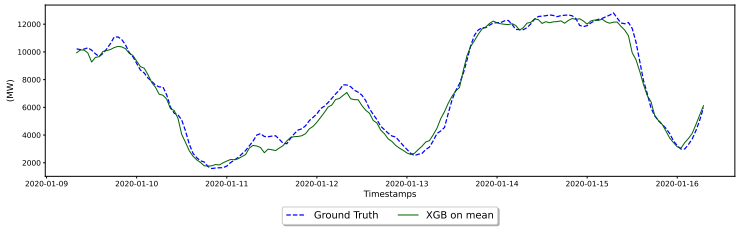


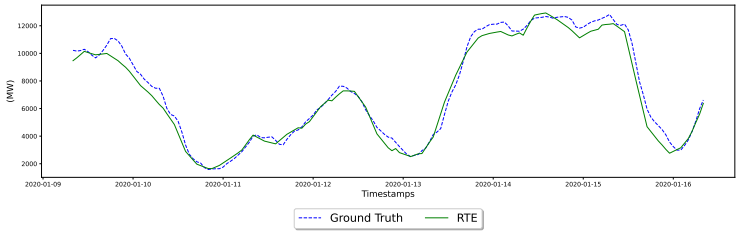
Figure D.5: Architecture found by WindDragon on Occitanie.



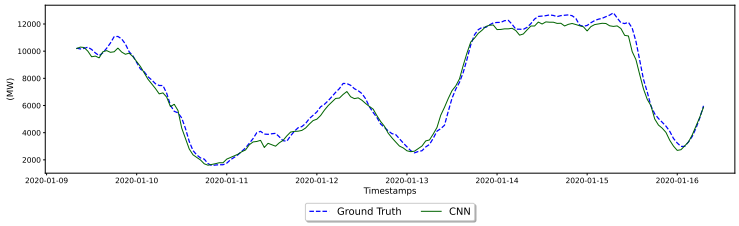
(a) Persistence forecast



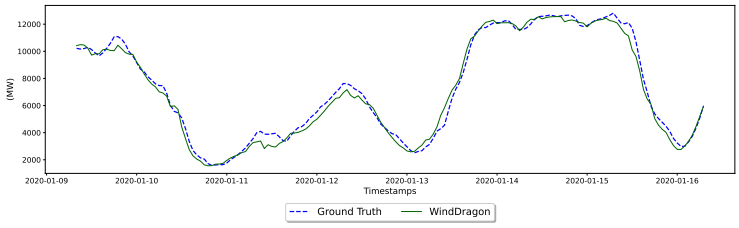
(b) XGB on mean forecast



(c) RTE forecast



(d) Convolutional Neural Network forecast



(e) WindDragon forecast

Figure D.6: Weekly comparative visuals