POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Machine Learning for Predicting Renewable Energy Generation

Supervisors

Prof. Michela MEO

Ph.D. Greta VALLERO

Prof. Danilo ARDAGNA

Candidate

Daniele CAPPUCCIO

Academic Year 2020-2021

Summary

In the last few decades, the world has faced a plethora of challenges related to energy sustainability, availability, and security. Since the negative effects on the environment and the economy are too heavy, conventional energy sources are coming under huge political and economical pressure. These circumstances have led to an increased interest in developing alternative and more sustainable energy sources, like solar photovoltaic, solar thermal, geothermal, tidal waves, wind power, and biomass. New challenges related to these energy sources have recently motivated the use of machine learning algorithms to support better management of energy generation and consumption. The formulation of well informed energy policies that help to determine important parameters can be facilitated by forecasting both short and medium term demand, at least for a power grid with renewable energy sources contributing a considerable amount of energy supply. This work investigated the problem of predicting renewable energy generation focusing on wind and solar energy. Modern deep learning techniques have been applied, and they were tested and compared on different locations and time resolutions, hence to detect and understand any patterns behind the physical phenomenons.

Table of Contents

Chapter 1

Introduction

In the last few decades, the world has been faced with a plethora of challenges related to energy sustainability, availability, and security. Since the negative effects on the environment and the economy are too heavy, conventional energy sources — which include natural gas, crude oil, and coal — are coming under huge political (and subsequently, economical) pressure. Our present era of fossil-fueled economies, societies and civilizations has given rise to an anomalous and dangerous moment for contemporary humanity and our shared biosphere as well [1]. The accelerating trends of planetary warming evidenced through storms and ice melts,droughts and hunger, unrest and migration, increasingly compel a heightened sense of urgency regarding the need to rapidly end the age of fossil fuels. A growing consensus now views the transition to renewable energy systems, frequently understood as a process of fuel substitution, as a key strategy to address the climate crisis.

These circumstances have led to an increased interest in developing alternative and more sustainable energy sources, like solar photovoltaic, solar thermal, geothermal, tidal waves, wind power, and biomass [2]. In fact, it is worth to notice that many countries and companies have been seeking to diversify their energy mix by increasing the share of renewables. For instance, the share of renewable energy in gross final energy use in the EU has doubled since 2005. It reached 17.6% in 2017 and increased further to 18.0% in 2018, according to the early estimates from the European Environment Agency (EEA). Moreover, renewable energy now accounts for 30.7% of gross final electricity consumption, 19.5% of energy consumption for heating and cooling, and 7.6% of transport fuel consumption in the whole EU. Fig. 1.1 from the 2016 Renewable Energy Data Book shows how this increase in share similarly affected the US market.

The need for developing sustainable energy sources also calls for a more modern

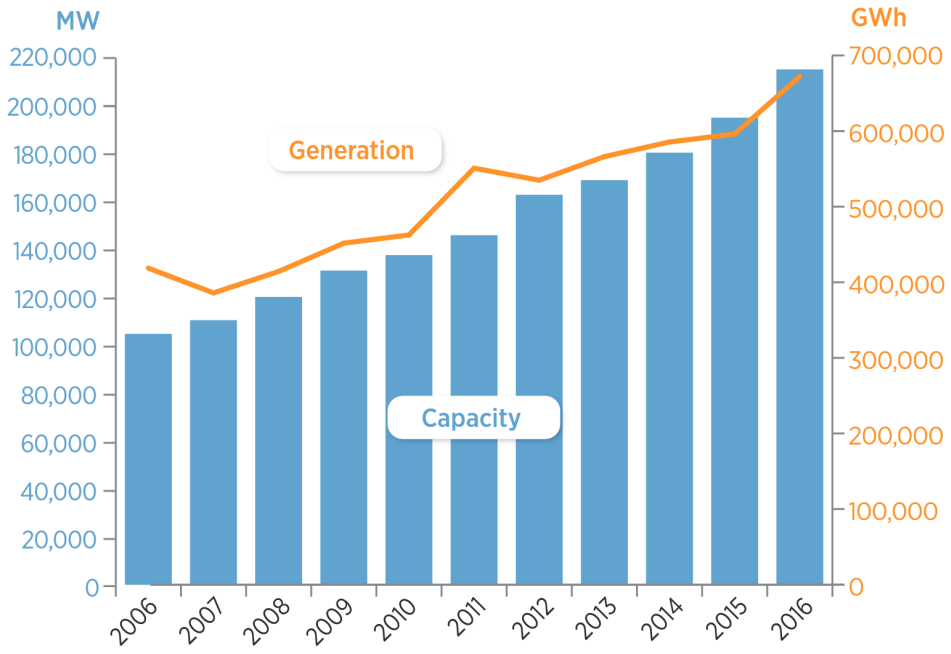


Figure 1.1: All renewables total nameplate capacity and generation in the United States over the years. Sources: EIA, LBNL, SEIA/GTM.

and efficient way of deploying this energy. In this sense, the definition of "Smart Grid" (SG) acquired more and more consensus and popularity. Indeed, many see SGs as an unprecedented opportunity to move the energy industry into a new era of reliability, availability, and efficiency that will contribute to our economic and environmental health. In short, a smart grid is defined in the TEN-E Regulation as an electricity network that can integrate in a cost efficient manner the behaviour and actions of all users connected to it, including generators, consumers and those that both generate and consume, in order to ensure an economically efficient and sustainable power system with low losses and high levels of quality, security of supply and safety.

In conventional energy generation processes, energy production depends on the energy demand from the users, and the stability of the power grid relies on the equilibrium of energy demand and supply. When this equilibrium is not achieved, a number of things can occur. For instance, when the demand is lower than the supply, energy is lost and wastage incurs unnecessary costs. On the other hand, when the energy demand surpasses the supply, the power grid becomes destabilized and this results in quality degradation and eventual blackouts in some parts of the grid. Thus, it is crucial to produce the right amount of energy at the right time,

both for the smooth running of the grid and for achieving economic benefits [3].

The main problem with renewable energy resources like wind, solar light, and solar heat is that they are highly variable in their fluctuations, so that the generation capacity can result in an instability in the power grid. Also, we cannot underestimate the fact the renewable energy power plants are subject to marked daily and annual cycles (e.g., solar energy is only available during the day). Consequently, it is necessary to generate power when resources are available and be able to store it for later use. In this scenario, conventional sources like gas plants are only employed to cover the electricity shortfall whenever the generation capacity of natural resources are insufficient to meet demand.

The aforementioned challenges have recently motivated the use of machine learning algorithms to support better management of energy generation and consumption. The formulation of well informed energy policies that help to determine important parameters (e.g., spinning reserve levels) can be facilitated by forecasting both short and medium term demand, at least for a power grid with renewable energy sources contributing a considerable amount of energy supply. It is also necessary to forecast the energy output from power plants, since this output depends on many environmental factors that cannot be controlled. Other areas for the application of machine learning methods in this context concern the overall operations and management of the smart grid — issues including fault detection, control, and so forth. These methods have also been used in determining the optimal location, size, and configuration of renewable power plants: indeed, factors like local climatic fluctuations, terrain, and proximity to population centers strongly affect this decision.

The objective of this thesis is to provide a machine learning framework based on neural networks and deep learning to predict the generation of energy from renewable sources, with an accent on wind and solar energy. Indeed, this prediction can be extremely useful to prevent instability in a SG, as we could potentially store eventual excesses for later use and employ them whenever the prediction yields a shortfall in meeting demand. In order to validate our models, we experiment with different geolocations, time resolutions, prediction horizons, and training approaches. Chapter 2 introduces the dataset used for this analysis and the data processing techniques. Chapter 3 focuses on neural networks and deep learning from a theoretical point of view, and how their tuning can have an impact on the prediction. Chapter 4 is the core of the thesis, in which all the achieved results are gathered and discussed. In the end, Chapter 5 follows the conclusions and possible applications.

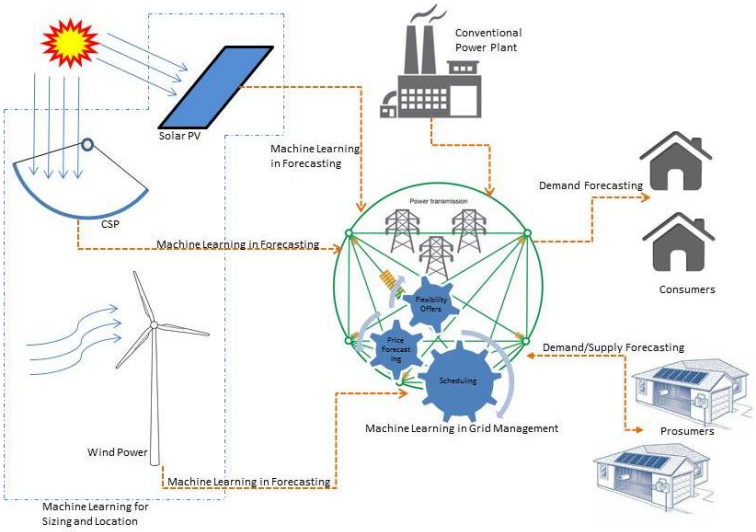


Figure 1.2: An overview of a power grid with integration of renewable sources and how machine learning techniques can be used at different steps of the process for performance improvements and overall better management. Conventional power plants are involved to guarantee the balance of demand and supply of energy and to ensure adequate power quality.

1.1 Machine Learning applications in literature

In research, a number of machine learning applications in energy-related scenarios have succeeded in achieving impactful results. For instance, ML has been applied to forecast traffic in the context of telecommunication networks — in particular, radio access networks [4]. The considered scenario comprises a network operation that was decided based on traffic. Energy generation predictions were derived from algorithms trained with past traffic and energy production patterns. The estimation of traffic represents a crucial step: if it is underestimated, the activation of base stations may lead to quality-of-service deterioration, while in case the traffic is overestimated, the energy saving would be suboptimal.

Other research has been focused on how energy efficiency techniques could apply to 5G networks [5]. By its nature, the 5G technology is remarkably different from the others in terms of reliability, network availability, latency, data rate, and so forth. The operational sustainability of such a diverse service support reckons on underlying network architecture that possesses a green nature, captured in its capability to well adapt the energy consumption to actual network traffic. For a network to be green, designers need to consider sustainability along a very broad

range of prospects. For example, we need to consider that even at zero load, network equipment still consume a fixed amount of energy, since that is used to guarantee that the network is, effectively, operational.

Cloud radio access networks (C-RANs) also provide a scenario in which energy cooperation techniques can be applied to improve the system performance with recent advances in smart grids. Long-term energy efficiency optimization can be achieved by considering both the instantaneous QoS and time-averaged limits. C-RANs are regarded as a key technical architecture for 5G networks, where base-band processing is operated in a centralized pool of units that handle requests from user equipment in a centralized and collaborative manner. Resource sharing in such cloud-based centralized pools has been shown to reduce power consumption, thus cutting down capital and operating expenditure. Compared with the traditional radio access network, C-RAN has more potential in boosting the transmission performance, whereas the coordination and allocation of radio resources has proven to be challenging [6].

Forecasting power output from a renewable energy power plant is crucial as this depends on many non-human-controllable factors such as environmental parameters. Depending on the energy source it uses, the power plant can exhibit various characteristics that enable the use of machine learning techniques for prediction purposes. In this thesis, we will focus solely on wind power generation prediction, although most of the employed methods can be easily generalized to work with other sources, like solar and hydro power.

Wind power generation depends on many characteristics and the power output from a wind turbine can be computed using Eq. 1.1, where A stands for the area that is covered by the wind turbine blades (a circle with radius r), ρ is the air density, V is the wind speed, and C_p is an efficiency factor usually imposed by the manufacturer.

$$P = \frac{1}{2} A \rho V^3 C_p \tag{1.1}$$

In this equation, the wind speed is a significant factor as the power output is proportional to the wind speed. It is also worth to notice that there is a cutoff speed where the power output is steady after that speed has been reached (so as to ensure the safety of the turbine). Other factors such as humidity and temperature also affect the density of the air, which in turn affects the power generation. Thus, it is necessary to forecast these factors and ultimately the final power output in a wind farm.

Wind forecasting plays an important role when it comes to clearing day ahead market scenarios. Given there is a market situation to be cleared, an accurate wind forecasting scheme is particularly helpful in such situations. Wind forecasting

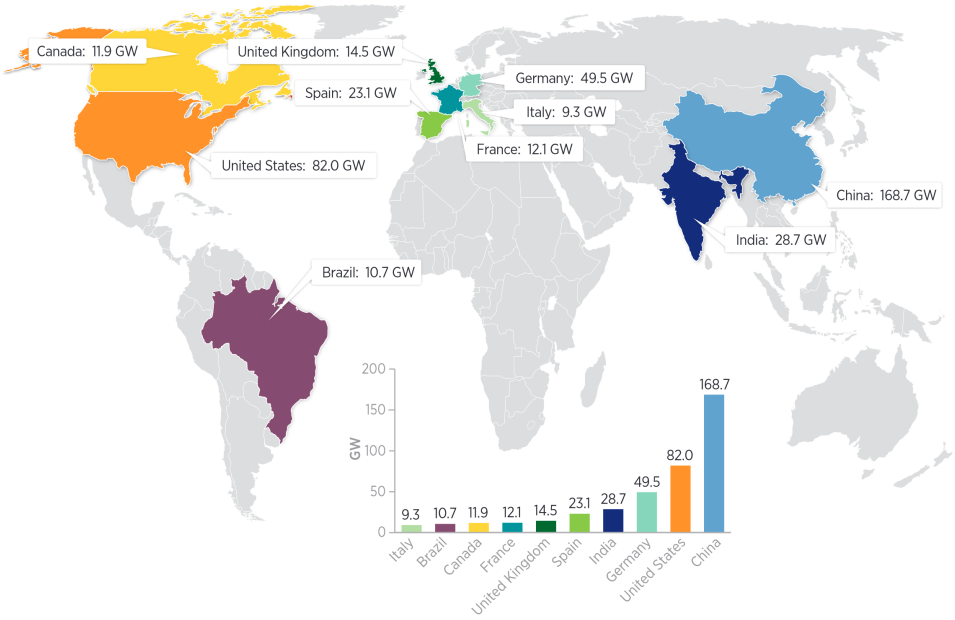


Figure 1.3: Top 10 countries by cumulative wind electricity capacity in 2016. Includes offshore wind. Sources: LBNL, REN21.

schemes can be roughly categorized in weather-based prediction methods and statistical-based prediction methods [7]. While we consider weather-based prediction models, the wind forecast accuracy strongly depends on the topology of the land where the wind turbines are erected. Wind speed measurements at an appropriate height from the land, the temperature of the ambient air, air pressure etc. hold important factors to take into account for the prediction. On the other hand, statistical (also knows as time series) methods solely depend on the past measurements of the wind to predict future values.

Wind forecasting applications lie majorly in the area of electricity market clearing, economic load dispatch and scheduling, and sometimes to provide ancillary support. Thus, a proper classification based on the prediction horizon — i.e. the duration of prediction — becomes important for various transmission system operators (TSOs).

In [8], experiments were conducted using an LSTM-based RNN to predict generated wind power in Sotavento, a wind farm in Spain. The main objective was to measure how the accuracy of the prediction changes when the prediction horizon enlarges. The accuracy of the models over the test set was validated using NRMSE. The authors observed that 1 hour-ahead forecasts produced a NRMSE of 4.23%, 3 hours-ahead 5.46%, up to a NRMSE value of 10.43% when predicting 24

hours-ahead.

Auto-regressive moving average (ARMA) and auto-regressive integrated moving average (ARIMA) models have been studied in [9] for wind speed forecasting and then wind power forecasting by analyzing the time-series data. Comparison of the ARIMA and artificial neural networks (ANNs) models for wind forecasting has been conducted in [10]. The analysis showed that the seasonal ARIMA model outperformed the ANN model, although limitations of the study have been identified in the poor number of training vectors.

A Kalman filter model using the wind speed as the state variable has been used in [11]. The authors suggested that this model is suitable for online forecasting of wind speed and generated power. Online forecasting of power generation can be crucial to provide the most recent and updated future forecasting to be used for power grid management. A recurrent multi-layer perceptron model which employs Kalman filter-based back-propagation has also been proposed in [12]. The proposed method performed well in long-term power generation prediction, while it failed in short-term prediction.

Chapter 2

Dataset and Data Processing

Energy system databases employ data methods to collect, clean, and republish energy-related datasets to be used for statistical analysis and for building numerical energy system models. In this chapter, we will focus on the Open Power System Data (OPSD) project database and describe data processing techniques to facilitate the analysis procedure.

2.1 Open Power System Data

The Open Power System Data project seeks to characterize the German and western European power plant fleets, their associated transmission network, and related information and to make the data available to energy modelers and analysts [13].

The original implementation of the platform is by the University of Flensburg (Berlin), the Technical University of Berlin, and the energy economics consultancy Neon Neue Energieökonomik, all from Germany. Developers collate and harmonize data from a range of government, regulatory, and industry sources throughout Europe. The project offers the following packages, for Germany and most other European countries:

- details, including geolocation, of conventional power plants and renewable energy power plants
- aggregated generation capacity by technology and country

- hourly time series covering electrical load, day-ahead electricity spot prices, and wind and solar resources
- NASA MERRA-2 satellite weather data
- electricity demand and self-generation time series for representative south German households
- simulated photovoltaic and wind generation capacity factor time series for Europe

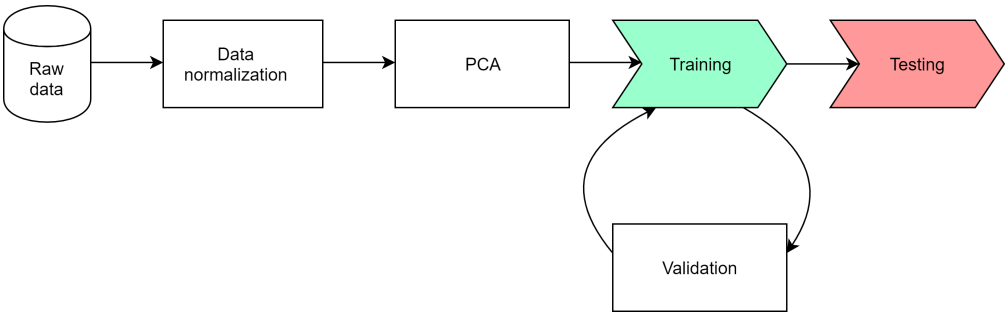


Figure 2.1: Procedural high-level schema. The first step after having acquired the raw data from OPSD is to normalize the data in order to reduce the intrinsic high variance and facilitate the training process of neural networks. We then perform Principal Component Analysis (PCA) to reduce the number of features of the dataset and project the data into a lower dimensional space, as to avoid the curse of dimensionality. Training and validation follows, during which our network "learns" from the data and is validated at the end of each epoch (see Sec. 3.6). Lastly, the testing phase concludes the process.

In order to facilitate analysis, the data is aggregated into large structured files (in `.csv` format), and loaded into data packages with standardized machine-readable metadata. The project also engages with energy data providers, such as transmission system operators (TSOs), and the European Network of Transmission System Operators (ENTSO-E). The data is available under an open license (Creative Commons), whereas the scripts deployed for data processing carry an MIT license.

In our analysis, we focus only on the time series package, as it contains different kinds of data relevant for power system modelling. The data is aggregated either by country (32 European countries are present), by control area or bidding zone. Where original data is available in higher resolution (half-hourly or quarter-hourly), like for Germany (DE), separate files are provided.

The package version we used is the 2019-06-05, which contains solely data provided by TSOs and power exchanges via ENTSO-E Transparency. Although

the package covers the period 2015-mid 2019, we only use the three-year span from Jan 1st, 2016 to Dec 31st, 2018, since the number of missing values is zero when we consider this period. Also, although the package also comprises solar power generation and capacities data, we will *not* consider them in our initial analysis — only in Sec. 4.4 the solar track will be used alongside wind in a hybrid scenario.

Table 2.1 shows the features of a sample dataset drawn from the hourly resolution time series with data belonging to Germany (DE), along with the respective timestamp (in UTC and CET time). The goal of our analysis it to try and predict the wind generation (target). It is worth to notice that the actual wind generation in megawatts is the sum of two components, namely the onshore and offshore actual wind generation. Fig. 2.2a and Fig. 2.2b show respectively the total wind energy generation in megawatts in January 2018 and the onshore and offshore components for the DE geographic zone.

2.2 Data processing

We normalized the dataset before feeding it to our learning algorithms in order to reduce the high internal variance, making data more regular. This transformation is often used as an alternative to zero mean, unit variance scaling. In particular, we transformed each feature individually such that it is scaled to a given range — in our case between 0 and 1 — using the following equation:

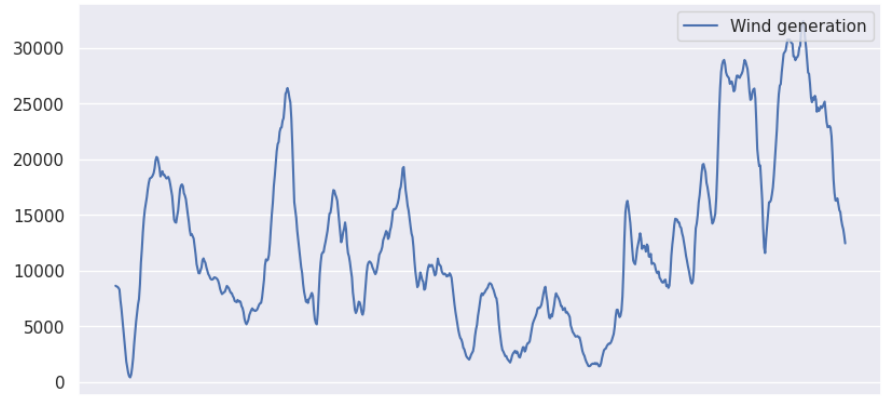
$$x_{std} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{2.1}$$

$$x_{scaled} = x_{std} \times (M - m) + m \tag{2.2}$$

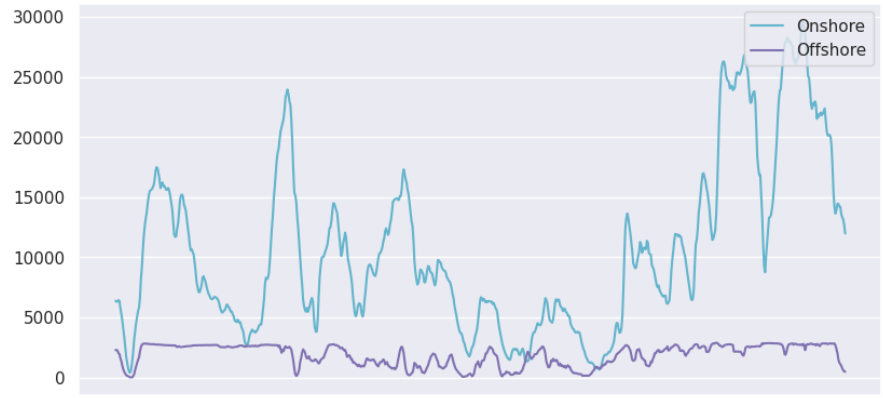
where x is the feature value, x_{max} and x_{min} are respectively the maximum and minimum value that feature occurs, and M and m represent the desired range of the transformed data (so we have $M = 1$ and $m = 0$).

It is crucial to normalize data before performing Principal Component Analysis (PCA), as explained in Sec. 2.3. If some variables have a large variance while others have a low one, performing PCA will load on the large variances. For example, if we scale one variable from “km” to “cm” (increasing its variance), it may go from having little impact to dominating the first principal component. Since we want PCA to be independent of such rescaling, normalizing the features will do that.

Fig. 2.3 shows the bivariate correlation matrix of the variables of our dataset. For obvious reasons, the timestamps were not included in the computation of the matrix. The Pearson correlation coefficient was used to compute the value of the



(a)



(b)

Figure 2.2: (a) Total wind generation in megawatts in Germany in January 2018 taken with hourly resolution. (b) As for (a), but onshore and offshore components are shown.

correlation. Values that are close to 1 reflect positive linear correlation, while 0 means no linear correlation between variables. For instance, we can see that the actual wind generation has a higher correlation with the onshore generation (0.99) than with the offshore generation (0.74), while having little to no linear correlation with the onshore (0.18) and the offshore (0.21) capacity.

2.3 Principal Component Analysis

Principal Component Analysis, or PCA, is a dimensionality-reduction method that is often used to reduce the dimensionality of large data sets, by transforming a

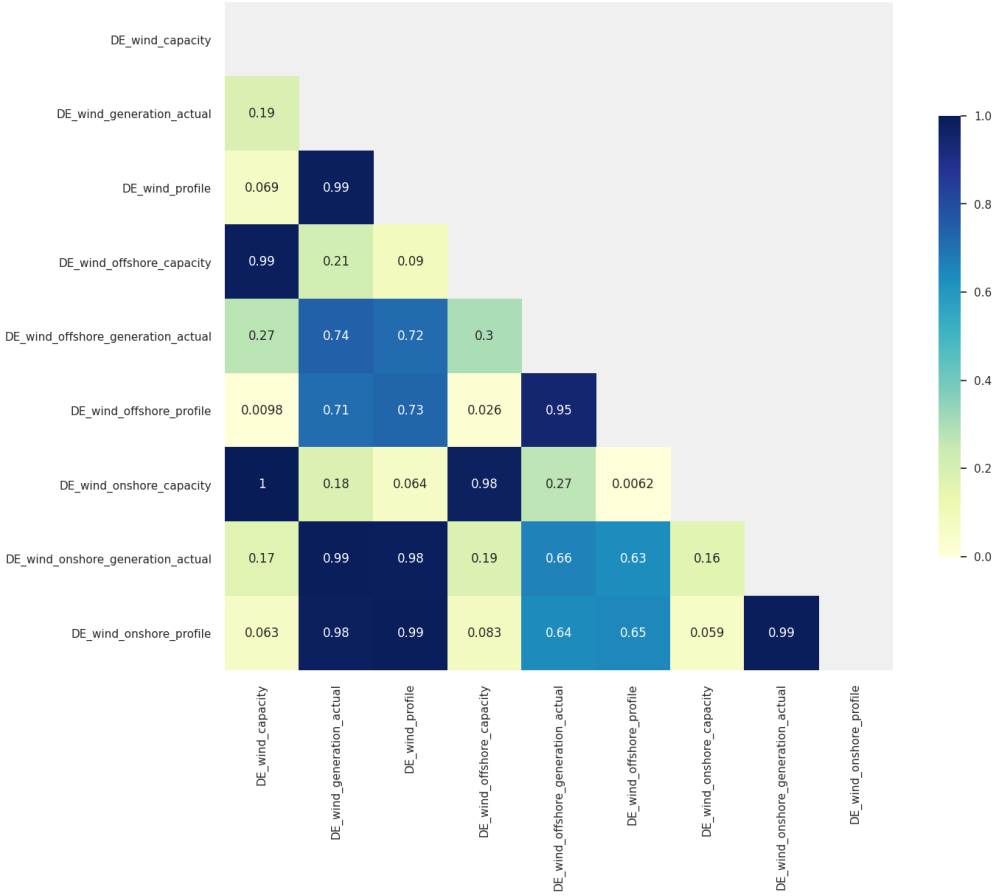


Figure 2.3: Correlation matrix heatmap between the features of the DE hourly resolution dataset. Some variable pairs share high correlation values, like the total wind generation and the onshore generation. This comes from the fact that onshore generation represents most of the total wind generation, as we can see from Fig. 2.2b.

large set of variables into a smaller one that still contains most of the information in the large set. Thus, the main goal of performing PCA is to reduce the number of variables of a dataset, while preserving as much information as possible.

Reducing the number of variables of a data set naturally comes at the expense of accuracy, but the trick in dimensionality reduction is to trade a little accuracy for simplicity. Since smaller data sets are easier to explore and visualize, this makes analyzing data much easier and faster for machine learning algorithms, without having to deal with redundancy and superfluous information. As for the number of significant components, it has to be determined, e.g. by cross-validation [14].

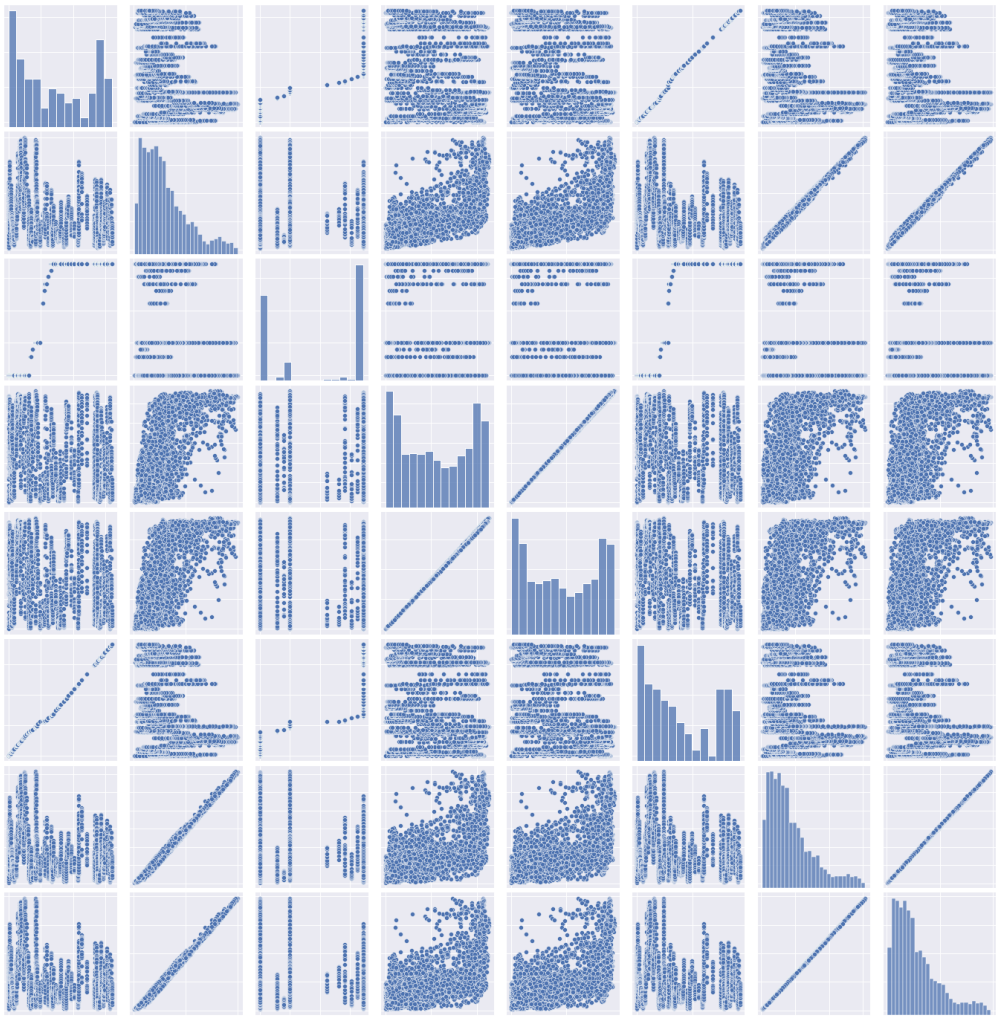


Figure 2.4: Plot showing pairwise relationships in the DE dataset. The order of the features is the same as in Fig. 2.3.

There are several ways of performing PCA on a given data matrix $A_{n\times d}$. The one discussed here addresses the use of a matrix factorization technique called the Singular Value Decomposition (or SVD) of A.

$$A = U\Sigma V^T \tag{2.3}$$

Here Σ is a $n \times p$ rectangular diagonal matrix (i.e., a matrix in which the entries outside the main diagonal are all zero) of positive numbers $\sigma_{(k)}$, called the *singular values* of A; U is a $n \times n$ matrix, and its columns are orthogonal unit vectors of length n called the *left singular vectors* of A; lastly, V is a $p \times p$ matrix whose

columns are orthogonal unit vectors of length p and called the *right singular vectors* of A.

In terms of this factorization, the matrix $A^T A$ can be written

$$A^T A = V\Sigma^T U^T U\Sigma V^T \tag{2.4}$$
$$= V\Sigma^T \Sigma V^T \tag{2.5}$$
$$= V\hat{\Sigma}^2 V^T \tag{2.6}$$

where $\hat{\Sigma}$ is the square diagonal matrix with the singular values of A and the excess zeros chopped off that satisfies

$$\hat{\Sigma}^2 = \Sigma^T \Sigma \tag{2.7}$$

The right singular vectors V of A are equivalent to the eigenvectors of $A^T A$, while the singular values $\sigma_{(k)}$ of A are equal to the square-root of the eigenvalues $\lambda_{(k)}$ of $A^T A$. Fig. 2.5 illustrates how the decomposition of the data matrix A via SVD is performed.

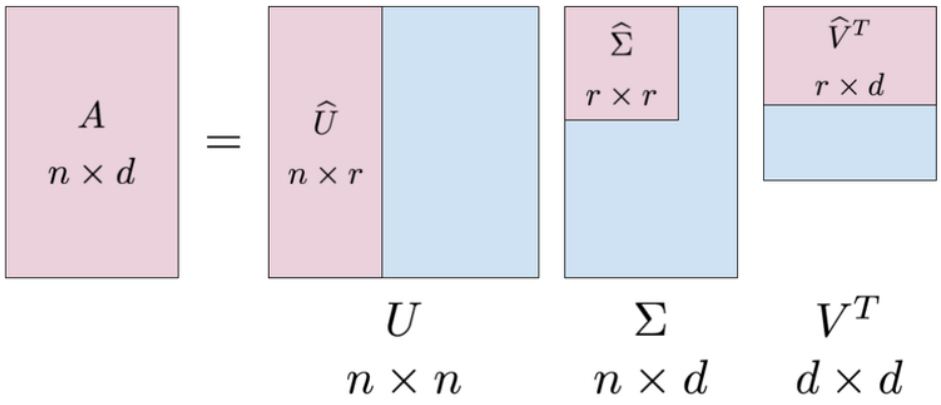


Figure 2.5: Schematic representation of the Singular Value Decomposition for a generic matrix $A_{n\times d}$.

As with other matrix factorization techniques, a truncated version of the SVD can be obtained by considering only the $L < p$ largest singular values and their respective singular vectors. In this way, we produce a truncated matrix that is the nearest possible matrix of rank L to the original matrix, such that the difference between the two has the smallest possible Frobenius norm.

Fig. 2.6 shows the explained variance ratio of each principal component selected when implementing SVD on the dataset. We can see that the first 3 components

already explain more than 80% of the total variance, with the first component accounting for nearly 50%. Although PCA may not be useful for every high-dimensional dataset, it still offers a straightforward and efficient path to gain insight into high-dimensional data and to understanding the intrinsic dimensionality of the data. In our case, we select 5 components — which explain 93% of the variance — as a good tradeoff to keep the number of components low (and avoid the "curse of dimensionality") while still holding on to most of the information held in the unprocessed dataset.

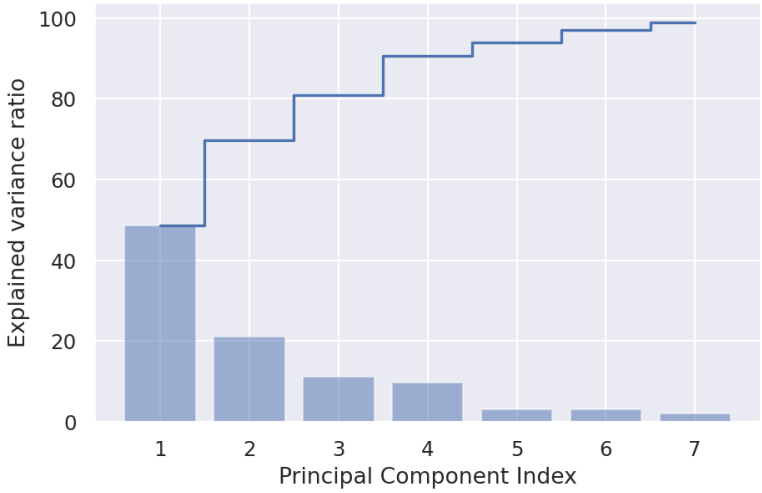


Figure 2.6: PCA Explained Variance Ratio, i.e., the percentage of variance that is attributed by each of the selected components. Ideally, we want to choose a number of components such that their explained variance ratio is above 80-90% to avoid overfitting.

When applied to time series, PCA is often called "Functional PCA" (or FPCA). Thus, FPCA refers to the situations when each of the n observations is a time series (i.e. a "function") observed at t time points. The point of FPCA is to find several eigenvectors of the covariance matrix, that would describe the "typical" shape of the observed time series. In [15], for instance, FPCA was used to investigate a high-dimensional surface water temperature dataset of Lake Victoria, both for univariate and bivariate functions, proving to be extremely efficient for dimensionality reduction as well as pattern detection.

In the following chapters, we assume that PCA has been performed on the data, that exactly 5 components have been selected, and that the explained variance ratio of these components is at least 90%.

Feature	Type	Description
utc_timestamp	datetime	Start of timeperiod in Coordinated Universal Time
cet_cest_timestamp	datetime	Start of timeperiod in Central European (Summer-) Time
DE_load_actual_entsoe_transparency	number	Total load in Germany in MW as published on ENTSO-E Transparency Platform
DE_load_forecast_entsoe_transparency	number	Day-ahead load forecast in Germany in MW as published on ENTSO-E Transparency Platform
DE_solar_capacity	number	Electrical capacity of solar in Germany in MW
DE_solar_generation_actual	number	Actual solar generation in Germany in MW
DE_solar_profile	number	Share of solar capacity producing in Germany
DE_wind_capacity	number	Electrical capacity of wind in Germany in MW
DE_wind_generation_actual	number	Actual wind generation in Germany in MW
DE_wind_profile	number	Share of wind capacity producing in Germany
DE_wind_offshore_capacity	number	Electrical capacity of wind_offshore in Germany in MW
DE_wind_offshore_generation_actual	number	Actual wind_offshore generation in Germany in MW
DE_wind_offshore_profile	number	Share of wind_offshore capacity producing in Germany
DE_wind_onshore_capacity	number	Electrical capacity of wind_onshore in Germany in MW
DE_wind_onshore_generation_actual	number	Actual wind_onshore generation in Germany in MW
DE_wind_onshore_profile	number	Share of wind_onshore capacity producing in Germany

Table 2.1: Feature description of the Open Power System Data time series for Germany.

Chapter 3

Neural networks

Artificial neural networks can be most adequately depicted as 'computational models' with particular properties such as the ability to adapt, learn, and generalize. Nonetheless, most of the aforementioned properties can be attributed also to non-neural models, so that the question is to which extent the neural approach proves to be suitable for certain applications than other models.

Neural networks fall in the category of *supervised learning* algorithms. This means that the neural network learns a function that maps an input to an output based on example input-output pairs. It infers a function from labeled training data consisting of a set of training examples (i.e., the "training set"). In supervised learning, each sample is a pair consisting of an input object (typically a vector) and a desired output value. Supervised learning algorithms analyze the training data and produce an inferred function, which can be used for mapping new examples. After this function has been inferred, the "test set" is used to assess the model's accuracy, according to specifics metrics. "Overfitting" is a common phenomenon that occurs in supervised learning algorithms when the model follows too closely the training data and fails at generalizing on new inputs, expressly performing poorly on the test set. Roughly speaking, overfitting is what we call when a model begins to "memorize" training data rather than "learning" to generalize from a trend.

A plethora of different architectures have been proposed over the last decades for neural networks. In our work, we focus on feedforward (FFNN) and recurrent (RNN) neural networks. In the former, better discussed in Sec. 3.2, the connections between the nodes of the graph defined by the network do not form a cycle. In the latter, extensions are added to the model to include feedback connections, i.e., outputs of the model are fed back into itself. RNNs are explained in Sec. 3.3.

If the supervised learning task is used to identify to which of a set of categories a new observation belongs, then we call that a “classification” task. Examples are assigning a given email to the "spam" or "non-spam" class (binary classification), or designating the proper species to a given plant (multi-class classification). On the other hand, if we have to predict a continuous variable, we call that task a “regression task”. For instance, predicting house prices is a regression task, since we want to estimate a number.

The process of estimating energy generation is clearly a regression task. The variable we want to estimate is also called “target”. For this purpose, and for the sake of clarity, we will use the following notation:

– y_i refers to the actual value the target feature takes at timestamp i;

– $\hat{y}_i$ is the output of our prediction model for the i-th sample, i.e., the predict value at timestamp i;

– $\bar{y}$ is the mean of the real values considering a time period of N timestamps.

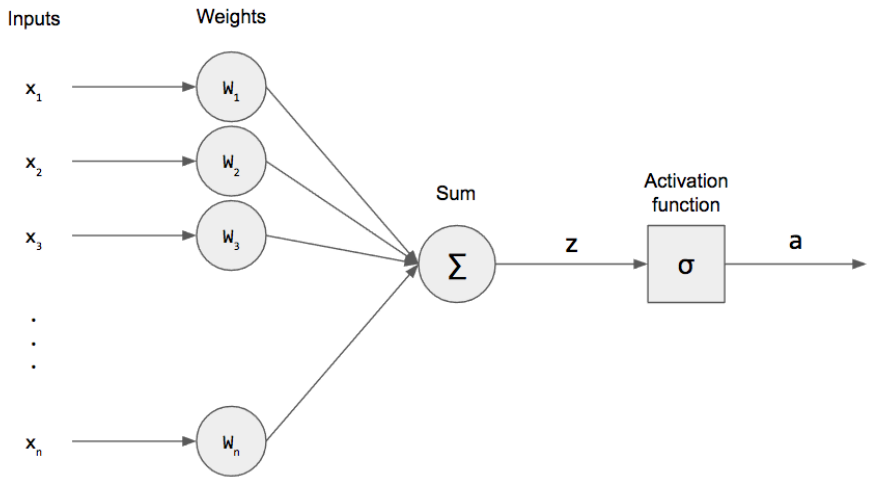


Figure 3.1: Schematic representation of the perceptron algorithm.

From a mathematical standpoint, an artificial neural network is a model capable of approximating any functional form with a certain level of precision. For instance, the first proposed artificial neuron — the so called *perceptron* [16] — takes several input signals $x_1, x_2, \ldots, x_n$ and summed weights them with $W_1, W_2, \ldots, W_n$ (the *weights*, i.e., real numbers indicating the importance of the different inputs). The

output (a) is a binary value: if the sum (z) is greater than a certain threshold (b), then (a) is 1, otherwise (a) is 0. (Fig. 3.1).

$$a = \begin{cases} 0, & \text{if } \sum_{i=1}^{n} x_i W_i \leq b \\ 1, & \text{if } \sum_{i=1}^{n} x_i W_i > b \end{cases} \tag{3.1}$$

The choice of the activation function σ is crucial for improving the accuracy of the neural network. A good characteristic for σ is the differentiability, which makes the backpropagation of the error during the training phase possible. Several activation functions have been used in research. A non-exhaustive list is reported below.

- Sigmoid (σ):

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.2}$$

  Historically, the sigmoid activation function has only been used in the output layers for binary classification problems, since it is not centred in zero and tends to saturate, causing the vanishing gradient problem.

- Hyperbolic tangent (tanh):

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3.3}$$

  Similar to the sigmoid activation function, but centred in zero. It still suffers from the vanishing gradient problem.

- Rectified Linear Unit (ReLU):

$$ReLU(x) = \max(0, x) \tag{3.4}$$

  ReLU is the most used activation function, as it does not saturate. However, it suffers from the 'dying ReLU' problem, i.e., the gradient is equal to zero for all negative values of x.

- Leaky ReLU (LReLU):

$$LReLU(x) = \max(\alpha x, x) \tag{3.5}$$

  Leaky ReLU is a variation of ReLU that does not suffer from the null gradient problem of ReLU. α is a parameter that usually takes small values, e.g., 1/10.

- Exponential Linear Unit (ELU):

$$ELU(x) = \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & otherwise \end{cases} \tag{3.6}$$

  ELU makes the mean activations closer to zero, which speeds up learning. It has been shown that ELUs can obtain higher classification accuracy than ReLUs.

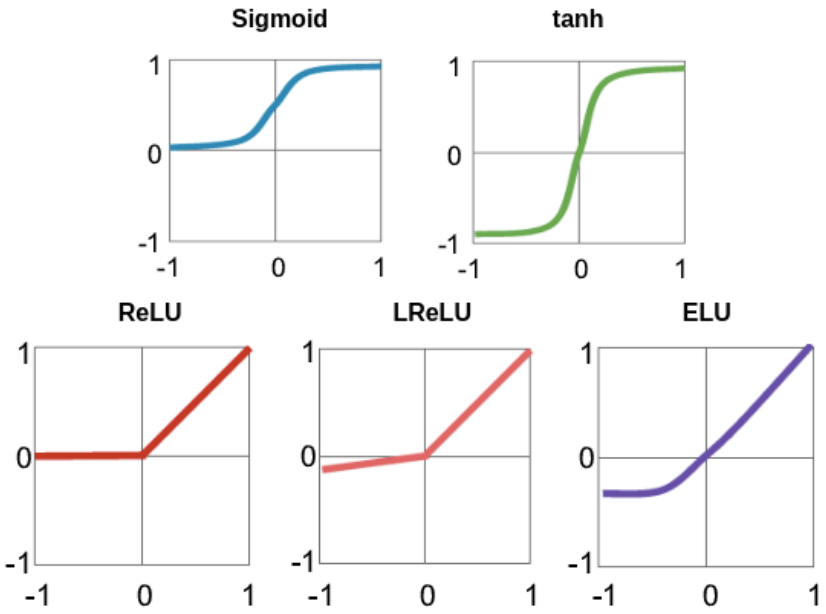


Figure 3.2: Graph visualization of activation functions.

The choice of the activation function is crucial for maximing the model's accuracy. Also, it has been shown that some activation functions perform better in some tasks, while failing in achieving the same results in other tasks. For instance, the sigmoid activation function is well suited for binary classification, although it performs poorly when it comes to regression. On the other hand, linear activation functions are usually the best alternative when it comes to regression.

3.1 Metrics

There are a variety of different metrics that are historically employed in a regression problem. A non-exhaustive list is reported below.

- Mean Squared Error (MSE):

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{3.7}$$

- Root Mean Squared Error (RMSE):

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2} \tag{3.8}$$

- Normalized Root Mean Squared Error (NRMSE):

$$NRMSE = \sqrt{\frac{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\bar{y}}} \tag{3.9}$$

- Mean Absolute Error (MAE):

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{3.10}$$

- Mean Absolute Percentage Error (MAPE):

$$MAPE = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \tag{3.11}$$

3.2 Feedforward neural networks

Feedforward neural networks (sometimes called *multilayer perceptrons*, in analogy with the perceptron) are the quintessential deep learning models. The goal of a feedforward network is to approximate some function f^*. For instance, if we have a classification task, $y = f^*(x)$ maps an input x to a category y. A feedforward network defines a mapping $y = f(x;)$ and learns the value of the parameters θ that result in the best function approximation.

As stated before, these models are called feedforward because information flows through the function being evaluated from x, through some intermediate computations, and finally to the output y. The connections between the nodes of the graph defined by the network do *not* form a cycle — there are no feedback connections in which outputs of the model are fed back into itself (see Fig. 3.3).

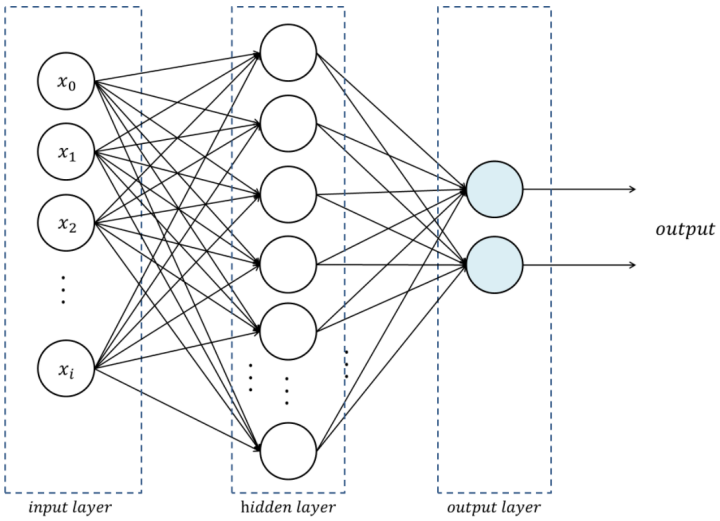


Figure 3.3: Schematic representation of a generic architecture of a feedforward neural network.

The first layer of a FFNN is the 'input layer'. No computation is involved with this layer, in fact, it just contains the inputs $x_1, x_2, \ldots, x_n$ of the neural network. Thus, the number of neurons in this layer matches the number of input features. Conversely, the last layer is called the 'output layer'. The number of neurons in this layer is determined by the number of elements of the output vector. For instance, in a general multiclass classification scenario, the number of neurons of the output layer is usually set equal to the number of classes involved in the classification.

Between the input and output layers stand a number of 'hidden layers', which also contain a number of neurons. Networks containing exactly one hidden layer are generally called 'shallow', in contrast to 'deep' networks, which contain more than one hidden layer. The number of layers and the number of neurons contained in each hidden layer represent important hyperparameters, since they can greatly affect the function approximation, as well the training process. Fig. 3.3 shows a schematic representation of the architecture of a feedforward network.

The training phase implies modifying the weights and the biases of the neurons as to minimize the error we make on the training samples that are fed into the network.

During training, the forward propagation of the input x to the hidden units up to the output layer produces a scalar cost. One of the most effective algorithms used for changing the weights of the neurons in a network in response to its output values is 'backpropagation' [17]. Often simply called *backprop*, it allows the information

from the cost to then flow backward through the network in order to compute the gradient.

Backprop uses the chain rule of calculus to compute the derivatives of functions formed by composing other functions whose derivatives are known. Once we have the error derivatives for one layer, we can use them to compute the error derivatives for the neurons in the previous layers.

For supervised learning, we choose a loss function L that is computed in terms of the target (the correct outputs of the training set), and the output vector of the network. Since the output vector depends on the input vector x and all the weights and biases of the neurons w, we can write:

$$L = L(y', y(x, w)) \tag{3.12}$$

In order to propagate the correction backwards, we need to compute the partial derivatives of the loss function with respect to the weights and biases:

$$\frac{\partial L(w)}{\partial w_{i,j}^k} \tag{3.13}$$

where k indicates the layer to which the neuron belongs, i refers to the number of the neuron of this layer (i.e., the one that receives the input), and j the number of the neuron of the previous layer whose output is the input to the current neuron. It is useful to introduce δ_j^k, that by definition is the *local gradient* of the jth neuron in the kth layer. This represents the partial derivative of the loss function with respect to the input z_j^k of the activation function of the considered neuron.

$$\delta_j^k := \frac{\partial L(w)}{\partial z_j^k} \tag{3.14}$$

The partial derivative can be easily computed using the chain rule. For instance, in the output layer:

$$\frac{\partial L_j(w)}{\partial w_{j,i}^{out}} = \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial w_{j,i}^{out}} \tag{3.15}$$

Backpropagation is a quite simple algorithm, although for neural networks with many neurons and layers it may become computationally expensive. There are many techniques that have been employed to make the learning process faster, like stochastic gradient descent and adaptive learning rate.

Weight initialisation is also crucial for effectively training FFNNs: the initial weights must break the symmetry of the system. In fact, if two hidden neurons with

the same activation function are connected to the same input they are updated accordingly, so that redundancy is introduced in the system.

Heuristics suggest that it is convenient to initialise the weights randomly with a distribution centred in zero. The most common distributions adopted for this purpose are the uniform distribution or the Gaussian distribution.

3.3 Recurrent neural networks

Recurrent neural networks are a family of neural networks used for processing sequential data. RNNs ccan scale to longer sequences than would be practical for networks without sequence-based specialization, like FFNNs. Also, the majority of recurrent networks are able to process sequences of variable length.

Within our work, we use RNNs to operate on sequences that contain vectors $x^{(t)}$ with the time step index t ranging from 1 to τ. The time step index reflects the passage of time in the real world (and not only the position in the sequence). The main idea behind the development of RNNs is to include cycles in the computational graph. These cycles represent the influence of the past values of the time series on its own value at the current time step.

A computational graph is a formal way of defining the structure of a set of computations, such as the ones involved in mapping inputs and parameters to outputs and loss in FFNNs. The intuition of chaining events and/or data into a computational graph that has a repetitive structure is called *unfolding*.

For instance, we consider the classical form of a dynamical system:

$$s^{(t)} = f(s^{(t-1)}; \theta) \tag{3.16}$$

where $s^{(t)}$ is called the state of the system. We say that Eq. 3.16 is recurrent because the definition of s at a certain timestamp t depends on the same definition at time $t - 1$.

For a finite number of steps τ, we can unfold the graph by applying the definition exactly $t - 1$ times. For instance, for $\tau = 3$ time steps:

$$s^{(3)} = f(s^{(2)}; \theta) \tag{3.17}$$

$$= f(f(s^{(1)}; \theta); \theta) \tag{3.18}$$

By repeatedly unfolding the equation using the definition, we have yielded an expression that does not involve recurrence. The expression can be represented by a traditional directed acyclic graph, as illustrated in Fig. 3.4.

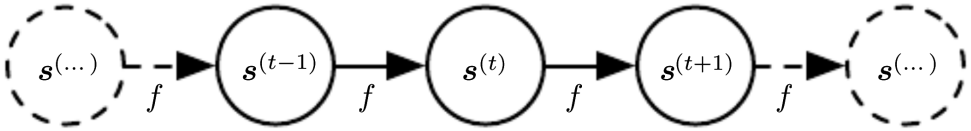


Figure 3.4: The classical dynamical system described by Eq. 3.16 illustrated as an unfolded computational graph. The function f maps the state at time t to the state at $t + 1$.

Now we consider the more general case of a dynamical system driven by an external signal $x^{(t)}$. Usually, recurrent neural networks use a variable h to compute the values of their hidden units, which represent their state. It is computed as:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta) \tag{3.19}$$

While the RNN is trained to predict the future state from the past, the network typically learns to use $h^{(t)}$ as a lossy summary of the task-relevant aspects of the past sequence of inputs up to t. Depending on the training criterion, the summary might selectively keep some aspects of the past sequence and discard others. For instance, in the context of language modelling, in order to predict the next word given previous words, it may not be necessary to store all of the information in the input sequence, but rather only enough information to predict the rest of the sequence. Fig. 3.5 depicts a recurrent network that processes information from the input x by incorporating it into the state h that is passed through time.

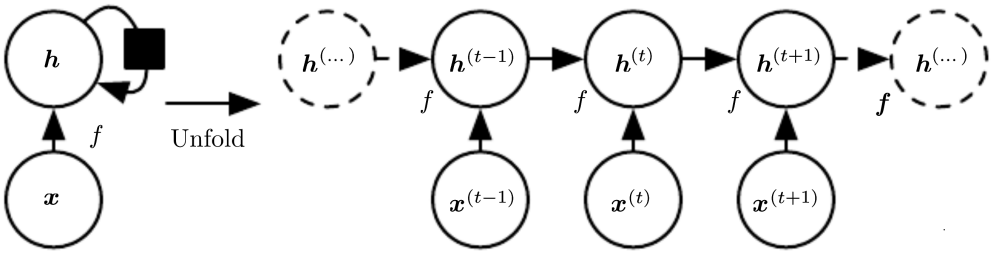


Figure 3.5: *(Left)* Circuit diagram of a recurrent network with no outputs, where the black square indicates a delay of 1 time step. *(Right)* The same network seen as an unfolded computational graph. Each node is associated with a single time instance.

3.3.1 Long Short-Term Memory

One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in "the clouds are in the *sky*" we don't need any further context – it's pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.

But there are also cases where we need more context. If we try to predict the last word in the text "I grew up in France, so I speak fluent *French*". Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large. Unfortunately, as that gap grows, RNNs become unable to learn to connect the information. The problem has been explored in depth in [18] and in [19].

Long Short-Term Memory networks — usually called "LSTMs" — are a special kind of RNNs that are capable of learning long-term dependencies, as they are explicitly designed to avoid the long-term dependency problem. Introduced by Hochreiter and Schmidhuber (1997) [20], they were refined and popularized by many people in following works. They work tremendously well on a large variety of problems, and are now widely used.

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, as we can see in Fig. 3.6a, this repeating module will have a very simple structure, such as a single **tanh** layer. The key to LSTMs is the cell state, the horizontal line running through the top of the diagram in Fig. 3.6b. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged. The LSTM module does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed of a sigmoid neural net layer and a pointwise multiplication operation.

First, we have the "forget gate". This gate decides what information should be thrown away or kept. Information from the previous hidden state — that contains information on previous inputs — and information from the current input is passed

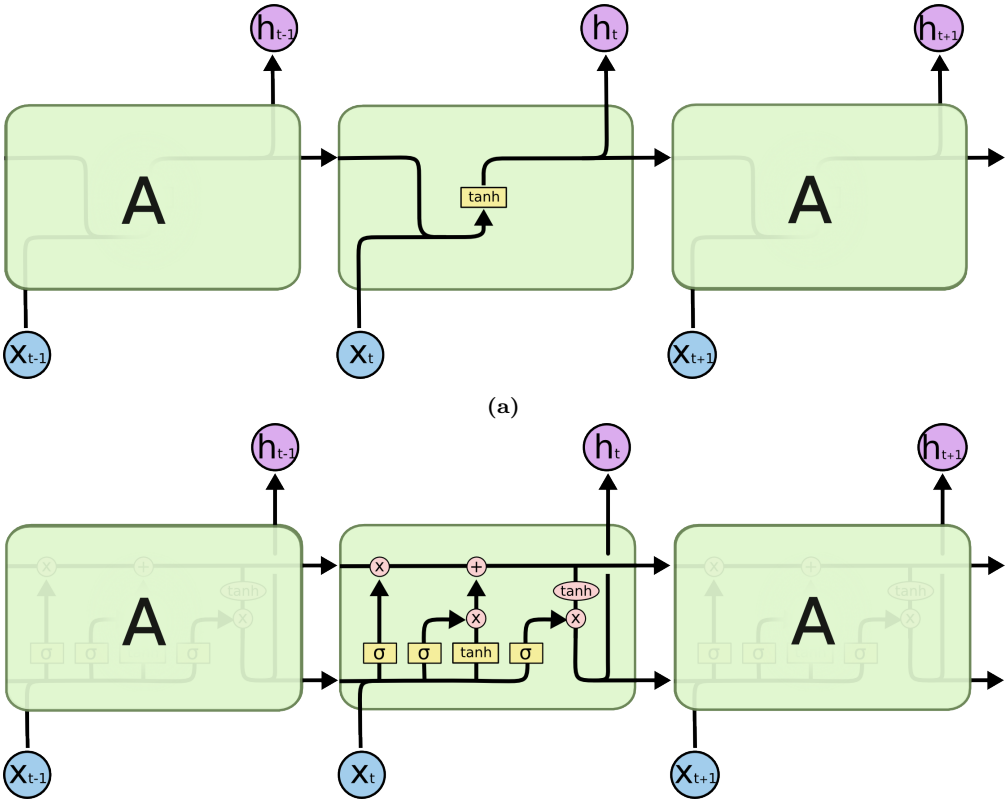


Figure 3.6: (a) The repeating module in a standard RNN, which usually contains a single layer. (b) The repeating module in an LSTM, which contains four interacting layers.

through the sigmoid function. Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep.

To update the cell state, we have the "input gate". First, we pass the previous hidden state and current input into a sigmoid function. That decides which values are updated by transforming the values to be between 0 and 1, where '0' means not important, and '1' means important. The cell state gets pointwise multiplied by the forget vector. This has a possibility of dropping values in the cell state if it gets multiplied by values near 0. Then, we take the output from the input gate and do a pointwise addition which updates the cell state to new values that give us our new cell state.

Last we have the "output gate". The output gate decides what the next hidden state should be. First, we pass the previous hidden state and the current input into a sigmoid function. Then we pass the newly modified cell state to the tanh

function. We multiply the tanh output with the sigmoid output to decide what information the hidden state should carry. The new cell state and the new hidden is then carried over to the next time step.

3.4 Gradient-based optimization

Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks. Its main usage is in the training phase of neural networks, where it is employed to minimize the loss function, i.e., the function in charge of evaluating a candidate model. In general, gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in \mathbb{R}^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t. to the parameters. Indeed, we can interpret the gradient vector as the "direction" of fastest increase of the loss function. Since we want to minimize the loss function, the "-" sign will do that (as in Eq. 3.20). The learning rate η determines the size of the steps we take to reach a (local) minimum. In other words, we follow the direction of the slope of the surface created by the objective function (the loss function) downhill until we reach a valley.

There are several variants of gradient descent, which differ in how much data we use to compute the gradient of the objective function. Depending on the amount of data, we make a tradeoff between the accuracy of the parameter update and the time it takes to perform an update.

Vanilla gradient descent, also known as *batch gradient descent*, computes the gradient of the cost function with respect to the parameters θ of the network (weights and biases) for the entire training set:

$$\theta = \theta - \eta \nabla_\theta J(\theta) \tag{3.20}$$

As we need to calculate the gradients for the whole dataset to perform just one update, batch gradient descent can be very slow and is intractable for datasets that don't fit in memory. Also, batch gradient descent does not allow us to update our model *online*, i.e., with new examples on-the-fly. Using this method, we update our parameters in the opposite direction of the gradients with the learning rate determining how big of an update we perform. Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.

Stochastic Gradient Descent (SGD) in contrast performs a parameter update

for each training sample $x^{(i)}$ and target $y(i)$:

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{3.21}$$

Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online. While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the other hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.

Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training samples:

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{3.22}$$

Despite this, vanilla mini-batch gradient descent, however, does not guarantee good convergence, but offers a few challenges that need to be addressed. These include, for instance:

- Choosing a proper learning rate. A learning rate that is too small leads to painfully slow convergence, while a learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.
- Deciding a good learning rate schedule by e.g. annealing, thus reducing the learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a threshold. These schedules and thresholds, however, have to be defined in advance and are thus unable to adapt to a dataset's characteristics.
- The same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.

To overcome these problems, a bunch of gradient descent optimization algorithms have been proposed in recent years. Among these algorithms, Adam [21] has stood out for being extremely powerful in optimizing deep neural networks.

Adam is an adaptive learning rate method, which means, it computes individual learning rates for different parameters. Its name is derived from adaptive moment estimation, and the reason it's called that is because Adam uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network.

Indeed, Adam computes two estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients. Since they are initialized as vectors of 0s (then biased towards zero), we counteract the bias by computing bias-corrected first and second moment estimates. We finally use these two values to update the parameters θ.

Alg. 1 portraits the details of the Adam optimization algorithm. As proposed by the authors of Adam, default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ are used. These values empirically showed how Adam compares favorably to other adaptive learning algorithms.

3.5 Regularization strategies

A central problem in machine learning is how to make an algorithm that performs well on the training data as well as the testing data — namely new inputs. Many techniques known collectively as 'regularization' are designed to reduce the test error (i.e., the error on the test set) possibly at the expense of increased training error (i.e., the error on the training set). Thus, we define regularization as any modification to a learning algorithm that is intended to reduce its generalization error and to prevent overfitting.

There are a lot of different regularization strategies. Some put extra constraints on a machine learning model, such as adding restrictions on the parameter values. If chosen carefully, these extra constraints and penalties can lead to improve performance on the test set. Sometimes these constraints and penalties are introduced to encode specific kinds of prior knowledge, or to express a generic preference for a simpler model class in order to promote generalization. Other forms of regularization, known as ensemble methods, combine multiple hypotheses that explain the training data.

In the context of deep learning and neural networks, most regularization strategies are based on regularizing estimators, so that it works by trading increased bias for reduced variance. What this means is that controlling the complexity of the model is not a simple matter of finding the model of the right size, with the right number of parameters, but of finding that the best fitting model — in the sense

Algorithm 1 Adam optimizer algorithm. All operations are element-wise, even powers.

procedure ADAM($\alpha, \beta_1, \beta_2, J, \theta_0$)
  ▷ α is the stepsize
  ▷ $\beta_1, \beta_2 \in [0, 1)$ are the exponential decay rates for the moment estimates
  ▷ $J(\theta)$ is the objective function to optimize
  ▷ θ_0 is the initial vector of parameters which will be optimized
  ▷ Initialization
  $m_0 \leftarrow 0$ ▷ First moment estimate vector set to 0
  $v_0 \leftarrow 0$ ▷ Second moment estimate vector set to 0
  $t \leftarrow 0$ ▷ Timestep set to 0
  ▷ Execution
  while θ_t not converged **do**
    $t \leftarrow t + 1$ ▷ Update timestep
    ▷ Gradients are computed w.r.t the parameters to optimize
    ▷ using the value of the objective function
    ▷ at the previous timestep
    $g_t \leftarrow \nabla_\theta J(\theta_{t-1})$
    ▷ Update of first-moment and second-moment estimates using
    ▷ previous value and new gradients, biased
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
    ▷ Bias-correction of estimates
    $\hat{m}_t \leftarrow \dfrac{m_t}{1 - \beta_1^t}$
    $\hat{v}_t \leftarrow \dfrac{v_t}{1 - \beta_2^t}$
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \dfrac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$ ▷ Update parameters
  end while
  return θ_t ▷ Optimized parameters are returned
end procedure

of minimizing generalization error — is a large model that has been regularized appropriately.

3.5.1 Batch Normalization

Batch normalization is a first method of performing regularization. Specifically, batch norm is used to make the training process of artificial neural networks faster and more stable through normalization of the input layer by re-centering and re-scaling [22].

In fact, each layer of a neural network has inputs with a corresponding distribution, which is affected during the training process by the randomness in the parameter initialization and the input data. The effect of these sources of randomness on the distribution of the inputs to internal layers during training is described as *internal covariate shift*.

Batch norm was thus initially proposed to mitigate this phenomenon. During the training stage of networks, as the parameters of the preceding layers change, the distribution of inputs to the current layer changes accordingly, such that the current layer needs to constantly readjust to new distribution. This problem is especially severe for deep networks, because small changes in shallower hidden layers will be amplified as they propagate within the network, resulting in significant shift in deeper hidden layers. Therefore, the method of batch normalization is proposed to reduce these unwanted shifts to speed up training and to produce more reliable models.

If we use B to denote a mini-batch of size m of the entire training set, we can denote the empirical mean and variance of B as:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{3.23}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{3.24}$$

For a layer of the network with d-dimensional input, $x = (x^{(1)}, \ldots, x^{(d)})$, each dimension of its input is then normalized (i.e., re-centered and re-scaled) separately:

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{(k)^2} + \epsilon}} \tag{3.25}$$

where $k \in [1, d]$, $i \in [1, m]$, and $\mu_B^{(k)}$ and $\sigma_B^{(k)^2}$ are the per-dimension mean and variance, respectively.

ϵ is added in the denominator for numerical stability and is an arbitrarily small constant. The resulting normalized activation $\hat{x}^{(k)}$ have zero mean and unit variance, if ϵ is not taken into account. To restore the representation power of the network, a transformation step then follows as:

$$y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)} \tag{3.26}$$

where the parameters $\gamma^{(k)}$ and $\beta^{(k)}$ are subsequently learned in the optimization process.

3.5.2 Dropout

Dropout is a computationally inexpensive but powerful method of regularizing a broad family of models. Dropout has the effect of making the training process noisy, forcing nodes within a layer to take on more or less responsibility for the inputs, simulating a sparse activation from a given layer. Roughly speaking, dropout can be thought as a method of making bagging practical for ensembles of very large neural networks. Bagging involves training multiple models, and evaluating multiple models on each test sample.

Specifically, dropout trains the ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base network, as illustrated in Fig. 3.7. In modern neural networks, we can effectively remove a unit from a network by multiplying its output value by zero.

In order to learn with bagging, we define k different models, construct k different datasets by sampling from the training set with replacement, and then train model i on dataset i. Dropout aims to approximate this process, but with an exponentially large number of neural networks. Specifically, to train with dropout, we use a minibatch-based learning algorithm: each time we load a sample into a minibatch, we randomly sample a different binary mask to apply to all of the input and hidden units in the network. The mask for each individual unit is sampled independently from all of the others. We can tune the hyperparameter of the probability of sampling a mask value of one — although empirical evidence suggests that a value of 0.8 for input units and 0.5 for hidden units is most of the times a good choice [23].

Although the cost per-step of applying dropout to a specific model is negligible, it reduces the effective capacity of a model. Indeed, the optimal validation set error is typically lower when using dropout, but this comes at the cost of a much larger model and many more iterations of the training algorithm. For extremely large datasets, regularization confers little reduction in generalization error. In

these cases, the computational cost of using dropout may outweigh the benefit of regularization.

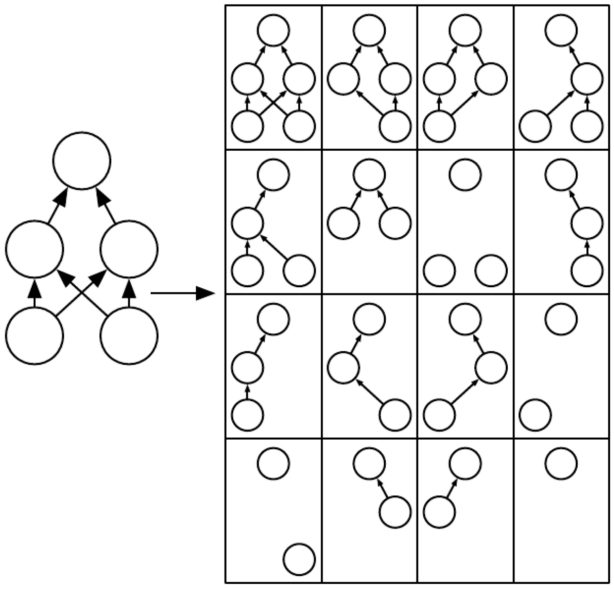


Figure 3.7: On the left, a sample network with 2 input neurons, 2 hidden neurons, and 1 output neuron. On the right, all the possible configurations of the same network after applying a binary mask to all non-output units.

3.6 Training approach

The training phase is by far the most crucial when it comes to minimizing the error in prediction, since even small changes in some hyperparameter can have an enormous impact on the performance of the trained model. For this purpose, we use validation to compute the error of the model on the validation set at the end of each epoch. In this section, we dig into the optimization of the training phase, as well as into the design choices that best suit our needs.

As we discussed in Sec. 3.2, a first important parameter of every neural network is the number of neuron units:

- **Input layer**. In the input layer, the choice of the number of neurons is quite straightforward. Since we applied a 5-component PCA to our dataset, we also

choose 5 units in this layer as to match the input vector size.

- **Hidden layers**. For selecting the number of hidden layers, as well as the number of neurons for each hidden layer, we trained models with different configurations and chose the one that minimized the MSE in the loss function. In the end, 3 hidden layers were selected, with the 1st and 3rd one containing 20 neurons each, and the 2nd one containing 40 neurons.
- **Output layer**. As for the output layer, in light of the fact that ours is a regression task which has to produce a single real number, we use only one neuron.

The FFNN-based model uses this very configuration, while the RNN-based only adds an LSTM module before the input layer. Batch normalization and dropout are added right after every non-output layer. LReLU with $\alpha = 0.1$ is used as an activation function. Moreover, the loss function employs the MSE metric to compute the error. Other important parameters include:

- the number of epochs, which was set to 100;
- the learning rate η, which was set to 10^{-4};
- the batch size, which was set to 128.

When not specified, we use the so called "yearly training", i.e., the models are trained and tested with data from all the months. In Sec. 4.1.4, we compare the yearly training with the "monthly training": in monthly training, we train 12 different models, each one on a different month, on a two-year span, and evaluate them on the same month in the test set, in order to see if any chance of improvement is there. In short, one model is fed with January data from the training set, and evaluated on the January data from the test set. This is repeated for each month, so 12 different predictors are to be trained and evaluated.

In the next chapter, we discuss the results obtained using the dataset described in Sec. 2 and the FFNN and RNN-based networks here defined, along with the chosen hyperparameters. The year 2016 is chosen as the split of the training set, while 2017 and 2018 serve respectively as the validation and test set. In Sec. 4.2, we explore the possibility of forecasting wind power generation at different prediction horizons, in order to determine how much difference in prediction accuracy we can get. As for the forecast approach, we implement the same strategies used in [24] to manage resources in a Base Station (BS) activation and deactivation, aiming at reducing energy consumption. For our prediction horizon, we use 6 hours. Thus,

we make use of the following approaches, using both a feedforward and a recurrent model:

- *1 NN-6 outputs.* A single neural network is used. At time t, the model outputs the prediction for wind power generation at time slots $t+1, \ldots, t+6$;
- *1 NN-1 output.* A single neural network is used. The model is trained to predict the generated power at the next time slot, i.e., $t+1$, and it is used in cascade to predict also the five future samples at time $t+2, \ldots, t+6$. This means that the neural network produces the prediction at time $t+1$, namely $\hat{y}_{t+1}$, using in input the generation at the previous time slot t, as well as the generation at times $t-1, t-2, \ldots$; once the value $\hat{y}_{t+1}$ has been computed, for predicting the power at time $t+2$, the same network is used but it receives as input the prediction $\hat{y}_{t+1}$ instead of the real value y_{t+1} that is unknown. Similarly, for predictions at time $t+3$ and $t+4$, predictions are used instead of the samples for the unknown values of the input;
- *6 NN-1 output.* Six different neural networks are used. Each network is dedicated to the prediction of the generated wind power at a given time lag. This means that the 6 future samples are separately predicted, using 6 different network, but the inputs are as in the previous case: predictions are used instead of missing samples whenever needed.

Chapter 4

Discussion of results

The following is a discussion of the results of our experiments made with the training approaches specified in Sec. 3.6. All the results, except for the ones described in Sec. 4.1.4, make use of the yearly training approach, i.e., a single predictor is trained over the entire training set. In Sec. 4.1.4, on the other hand, we make a comparison between the yearly and monthly approach. In the latter, a different predictor is trained for each month in the training set and evaluated on the corresponding month in the test set.

When not specified, we use data from Germany taken with hourly resolution from 2016 to 2018. Other data used in the following include the same data but with quarter-hourly resolution, and data from the United Kingdom taken with hourly granularity. The main difference between DE and UK data is the maximum and minimum values. In particular the maximum value registered in Germany is greater than twice the maximum value in the UK, while the median value in the latter is almost three times less than the median value in the former.

4.1 Base results

Once the design phase of our neural network models has ended, we test them using a variety of methods. First of all, we compare how FFNN and RNN perform on our base dataset (i.e., DE data with hourly resolution) using the metrics defined in Sec. 3.1 and figures to visualize the data. Then, we compare the results obtained by drawing the data from the same geographic location, but with different resolution (namely, hourly and quarter-hourly granularity). Similarly, we draw a comparison between countries, i.e., DE and UK. We also take into account how these results

can vary according to the month, so that a different prediction accuracy can be achieved by pointing out the month in which the prediction actually takes place.

4.1.1 Comparing FFNN and RNN

In this section, we make a simple comparison between the FFNN and RNN models by training them with our base dataset, i.e., DE data with hourly resolution (from Jan 1st, 2016 to Dec 31st, 2018). We train them using 2016 data, while we test them with the last year, 2018. The 2017 data is reserved as a validation set, so that it provides an unbiased evaluation of a model fit on the training dataset while tuning the model's hyperparameters.

Fig. 4.1 and Fig. 4.2 show respectively the training loss (MSE) for FFNN and RNN. Although we train them both for 100 epochs, additional tests were made using a larger number of epochs. Nonetheless, we did not notice a significant reduction in terms of training or validation loss, nor in accuracy on the test set for both models. We notice a slight difference between the two concerning the pace of training. Indeed, the RNN-based predictor is able to learn faster with respect to FFNN, although this difference becomes negligible as the epochs go by.

Both models keep their training and validation error more or less constant after a number of epochs, which is higher for the FFNN. Despite this, the order of error is the same — with FFNN showing a slightly larger difference between training and validation error in all epochs and a slightly larger validation error overall than RNN. For both models, the training error exceeds the validation error only in the first few iterations, while in further epochs the latter error is always lower than the former. Nevertheless, no increase in validation error was detected during each experiment, suggesting that no overfitting occurs and the early stopping technique did not impact the training of our models.

Fig. 4.3a and Fig. 4.4a show the predictions obtained with, respectively, the FFNN and RNN models in a 72-hour stretch, randomly selected in the part of the test set, which corresponds to January 2018. Also in this case, we see no significant difference between the two approaches. Both models tend to overestimate local maxima and underestimate local minima. Nevertheless, it is evident that RNN performs better than FFNN when these levels of production occur. This is undoubtedly due to the memory mechanism, which characterizes the RNN, explained in Sec. 3.3.1.

Overall, both models perform very well on the test set. Tab. 4.1 sums up the results in terms of MAE and MAPE achieved by FFNN and RNN, showing the variation of these errors in each month.

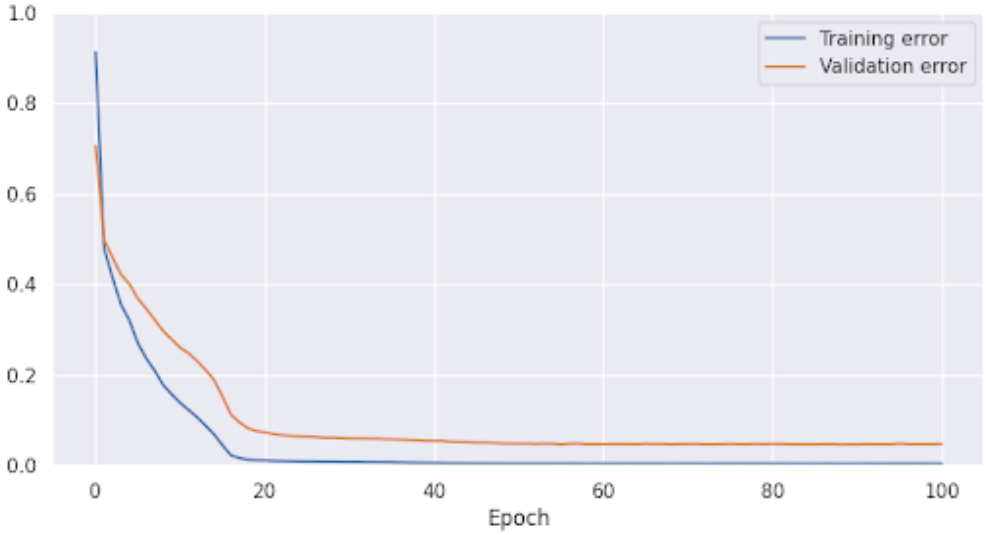


Figure 4.1: Training and validation loss (MSE) of the FFNN-based model, using DE data with hourly resolution.

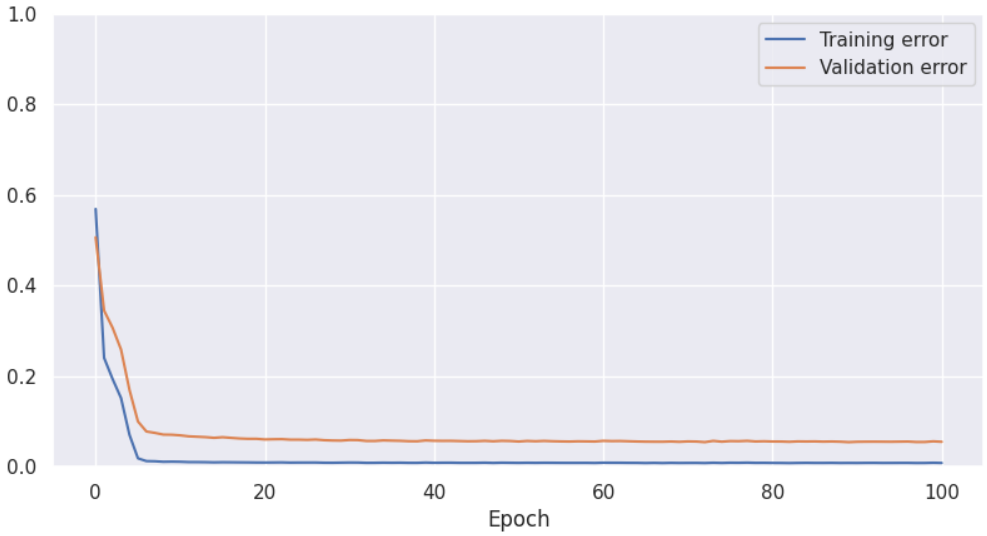
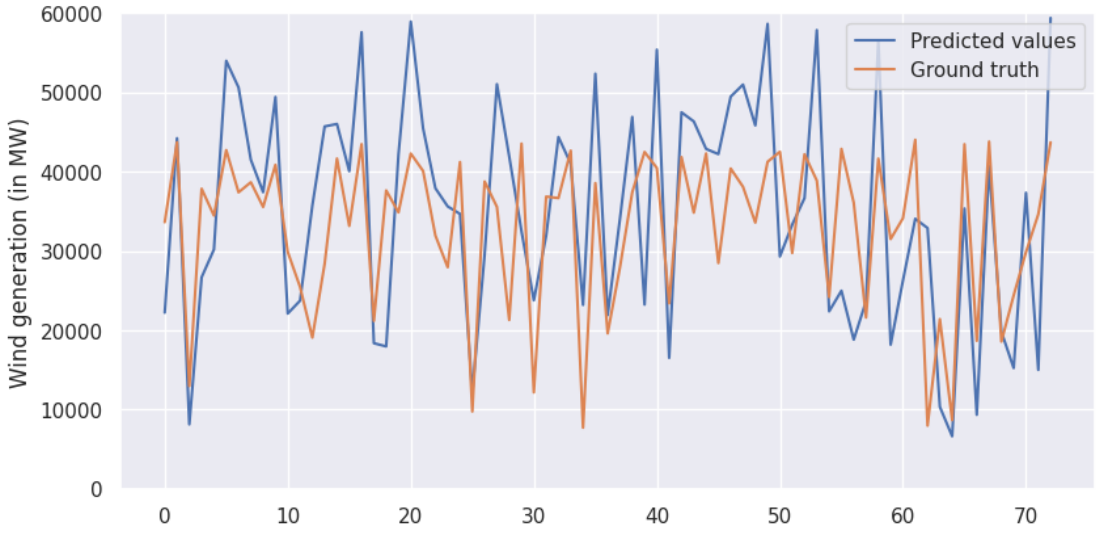
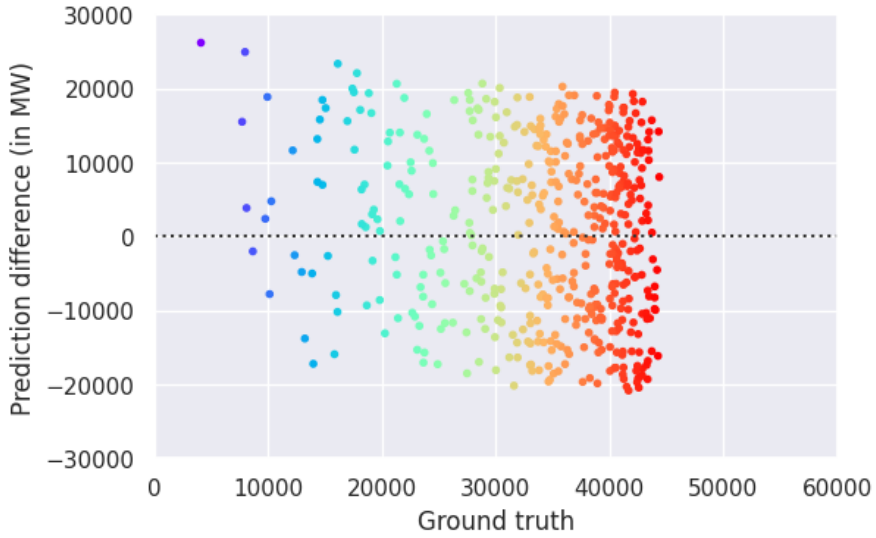


Figure 4.2: Training and validation loss (MSE) of the RNN-based model, using DE data with hourly resolution.



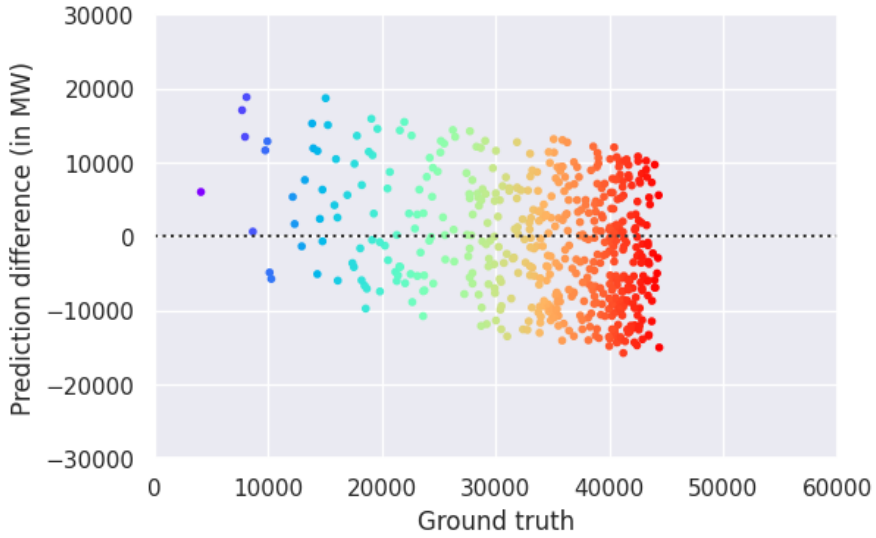
(a)



(b)

Figure 4.3: (a) FFNN prediction in actual wind generation compared to ground truth in a 72-hour stretch drawn from the test set (b) The resulting Bland-Altman plot considering all the test set.

(a)



(b)

Figure 4.4: (a) RNN prediction in actual wind generation compared to ground truth in a 72-hour stretch drawn from the test set (b) The resulting Bland-Altman plot considering all the test set.

	FFNN		**RNN**	
Month	*MAE*	*MAPE*	*MAE*	*MAPE*
January	8022.5	24.4	6970.4	19.8
February	7645.4	23.9	6259.6	19.1
March	7015.0	20.6	5534.2	17.2
April	7128.9	20.8	5698.4	17.6
May	5861.2	18.5	4997.5	15.0
June	6005.9	19.0	4295.5	13.2
July	4208.7	12.4	2987.0	10.4
August	4877.3	14.6	3073.8	11.2
September	5208.8	16.7	3833.9	13.4
October	5644.5	18.3	4932.6	15.8
November	6567.2	19.9	5201.5	16.7
December	7440.1	22.0	6187.5	19.4
	5984.2	**18.9**	**4566.1**	**14.2**

Table 4.1: Mean Absolute Error (MAE) and Mean Absolute Percentage Error (MAPE) by FFNN and RNN on the test set, with monthly granularity.

FFNN achieved an average MAE of 5984.2 and a MAPE of 18.9. RNN, on the other hand, achieved a MAE of 4566.1 and a MAPE of 14.2 — thus, the RNN model sees a decrease of the median absolute error of around 30%, which trades with the larger number of trainable parameters. Both FFNN and RNN perform around twice as better in the summer months with respect to the winter months, with a minimum MAPE of 10.4 in July (RNN) and a maximum MAPE of 24.4 in January (FFNN). Also, the median error for both FFNN and RNN is actually close to the MAE. For FFNN, the median error was 6015.6 (compared to a MAE of 5984.2), while for RNN it was 4377.1 (compared to 4566.1). This points out that the prediction is not biased towards an overestimation or an underestimation of wind power generation.

We can state that the results clearly show an advantage in using RNN with respect to FFNN, as we would expect in this kind of scenario. Nonetheless, we had to take into account the increase in the number of trainable parameters ($\approx 20\%$ in our case), which also leads to a slight increase in terms of training time (around 10%). Overall, we still consider using the RNN approach as the more suitable, especially in winter months when the difference in terms of MAE is larger than in summer5. Moreover, RNN seems to be more conservative when we have to predict very high or very low values of power, so that the error when an overestimation (or underestimation) takes place is slighly lower with respect to FFNN.

4.1.2 Comparing hourly and quarter-hourly resolutions

In this section, we compare the results we get using the dataset described in Sec. 4.1.1 and the same dataset but taken with quarter-hourly resolution. As for the model selection, we still implement both the feed-forward and the recurrent networks.

The choice of the time resolution — which, in our case, also matches the prediction horizon — can be critical in most systems, as different sampling rates can enable more or less accurate wind power forecasting. Although literature categorizes both hourly and quarter-hourly prediction horizons as *short-term* (opposed to *medium-term* and *long-term*, whose ranges span respectively from approximately 6 hours to 1 day, and from 1 day up to even a month), research [25] has suggested that even small changes in the metholodogy or the temporal resolutions can yield significant performance improvements.

We compare the train and validation losses obtained with quarter-hourly resolution with those with hourly resolution, with FFNN and RNN. Figs. 4.5 and 4.6 report the losses with FFNN and RNN methods, respectively, for the training (blue) and the validation (orange) sets. From these figures, it is evident that the curves are smoother and more regular, particularly in the first 5 to 10 epochs. Also, both the training and validation error at the end of the 100 iterations is slightly lower for both FFNN and RNN.

Fig. 4.7a and 4.8a show the predictions made using FFNN and RNN, respectively. For simplicity and comparison purposes, we still plot only a 72-hour stretch excluding intermediate values (i.e., contemplating only full hours). Although most of the aforementioned considerations are still valid in this experiment, we underline that the FFNN-based model trained with quarter-hourly resolution data achieved an interesting 16.3% MAPE (compared to 18.9). RNN also improved by a margin of almost 2% (14.2% down to 12.5%). This indicates that, although negligible, some improvements can be made if we are able to gather data with higher and higher resolution.

Last but not least, we want to point out that models trained with quarter-hourly resolution data perform fine when tested on hourly-resolution data. For instance, training an RNN-based model with quarter-hourly resolution data from 2016 and 2017 yields a MAPE of 12.7% when tested on 2018 data with hourly resolution. This means that higher-resolution data can help neural networks to capture more information and to detect patterns. The contrary is not necessarily true: in fact, when training an RNN with hourly-resolution data from 2016 and 2017 and testing it on quarter-hourly resolution data from 2018, we get a MAPE of 15.9% — hence, a slight deterioration in performance.

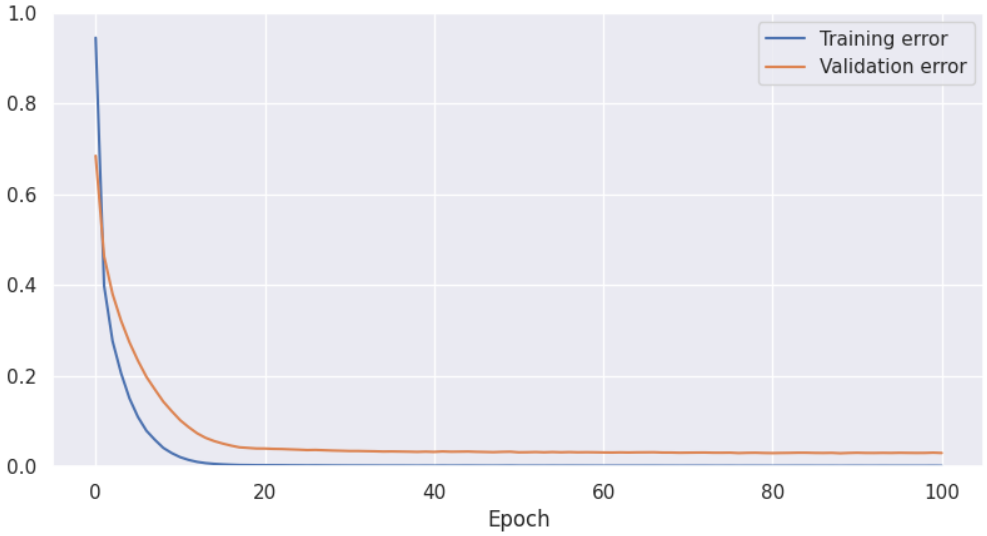


Figure 4.5: Training and validation loss (MSE) of the FFNN-based model, using DE data with quarter-hourly resolution.

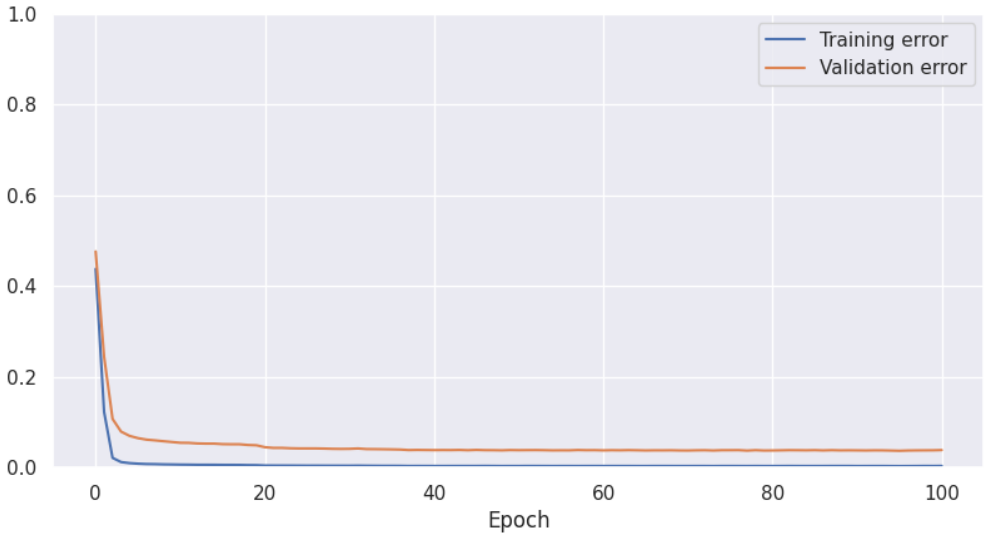
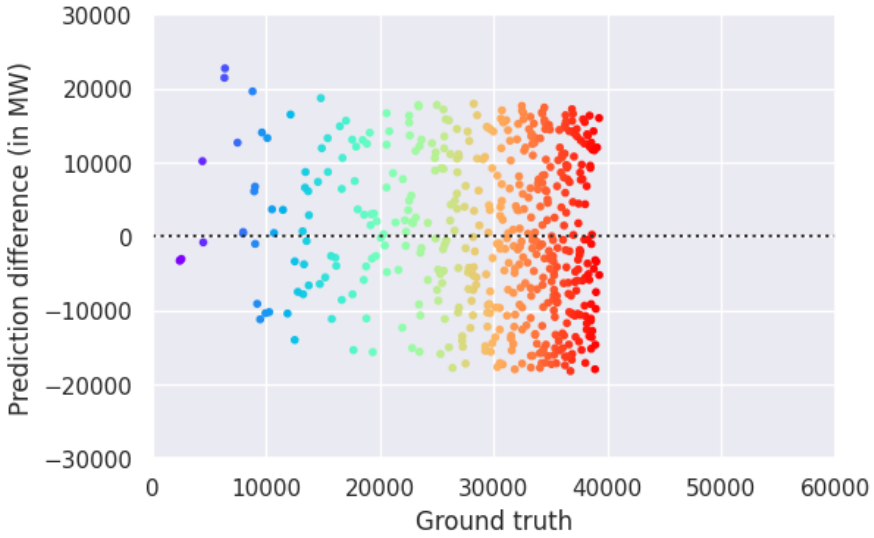


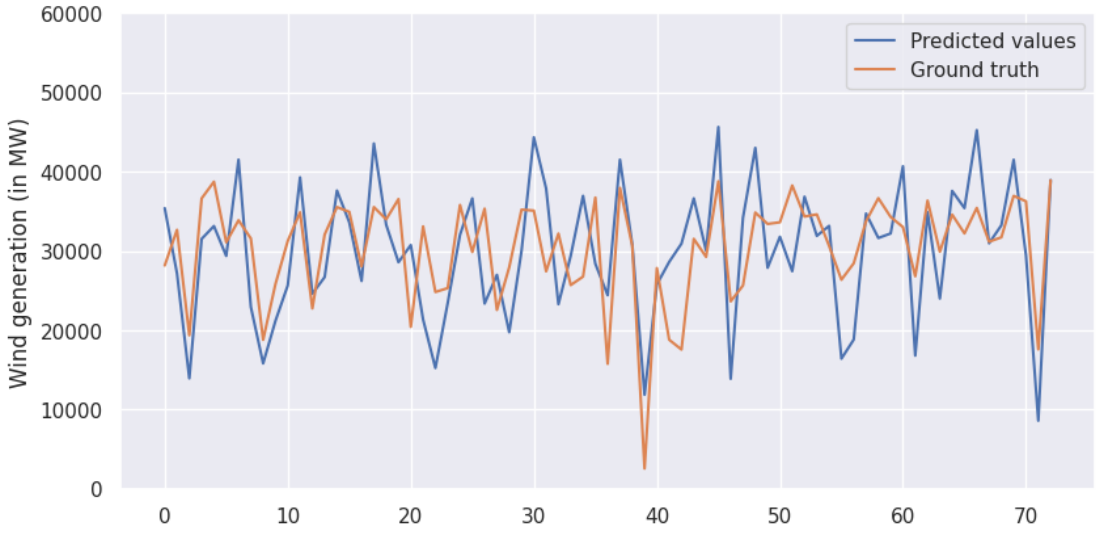
Figure 4.6: Training and validation loss (MSE) of the RNN-based model, using DE data with quarter-hourly resolution.

(a)

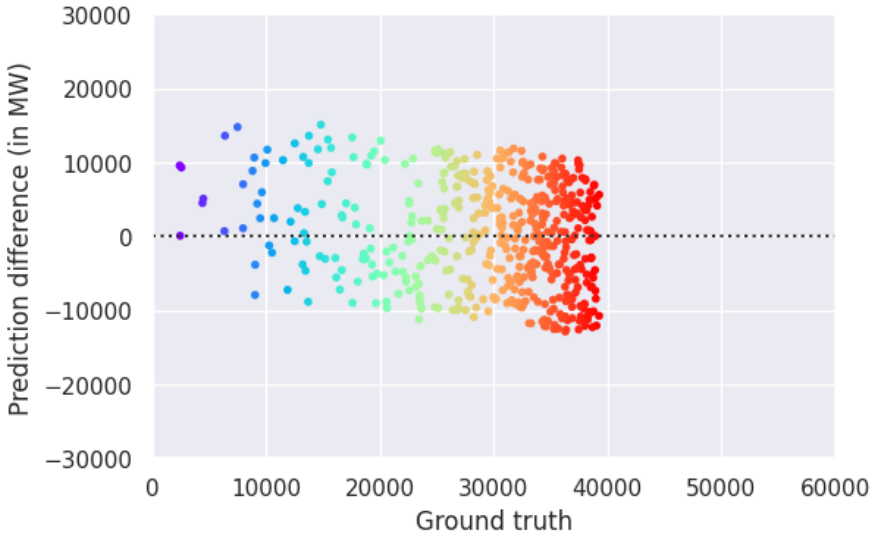


(b)

Figure 4.7: (a) FFNN prediction in actual wind generation compared to ground truth in a 72-hour stretch drawn from the test set, considering DE data with quarter-hourly resolution (b) The resulting Bland-Altman plot considering all the test set.



(a)



(b)

Figure 4.8: (a) RNN prediction in actual wind generation compared to ground truth in a 72-hour stretch drawn from the test set, considering DE data with quarter-hourly resolution (b) The resulting Bland-Altman plot considering all the test set.

4.1.3 Comparing Germany and United Kingdom

In this section, we want to test if the same architecture and training approach used in Sec. 4.1.1 to predict wind power generation in Germany can still yield effective results when we contemplate data with different characteristics, e.g., from different geographic locations. In this case, we gather wind data from the United Kingdom (UK) in the same exact time period (2016-2018).

The features are precisely the same as with DE data, and the preprocessing of the data is the same as well. As for the DE case, we normalize every feature into the range (0, 1), then we perform PCA with 5 components. In the UK scenario, these components explain 91% of the variance of the original dataset, so slightly less than the 93% we got with Germany. As explained at the beginning of this chapter, DE and UK data offer different characteristics, especially in the range of values, as well as the average and median value.

Fig. 4.9 and Fig. 4.10 show the training and validation losses for the FFNN and RNN, respectively. With respect to what we observed in Sec. 4.1.1 when comparing losses, in this case there is almost no difference in number of epochs to converge between the two models, while this difference was very noticeable with DE data. Also in this case, we have a slightly lower training and validation error with RNN, and no early stopping was required to avoid overfitting the data.

The predictions shown in Fig. 4.11a and in Fig. 4.12a confirm most of the statements we shared while dealing with DE data. The RNN model has an edge when it comes to predicting rapidly changing values, as well as in overall accuracy. Indeed, RNN achieved a MAPE of 15.5, while FFNN is less accurate with a MAPE of 18.7. This means that, with UK data, the difference between the accuracy of RNN and FFNN is less pronounced than the one we obtained with Germany (14.2 and 18.9). It is also worth to notice that the difference between winter and summer is less pronounced as well: for example, with RNN, we have a maximum MAPE of 18.2 in December and a minimum MAPE of 11.6 in July. In Germany, with the same model, the maximum and minimum values were respectively 19.8 and 10.4 (see Tab. 4.1). Overall, we can say that our model architectures are able to generalize well when fed with data with different geographic sources and that offer separate characteristics.

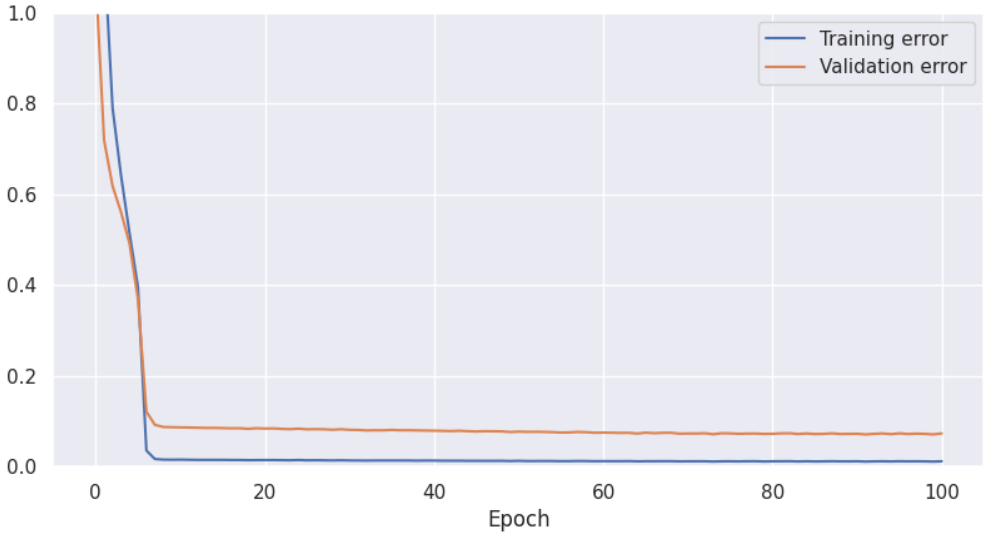


Figure 4.9: Training and validation loss (MSE) of the FFNN-based model, using UK data with hourly resolution.

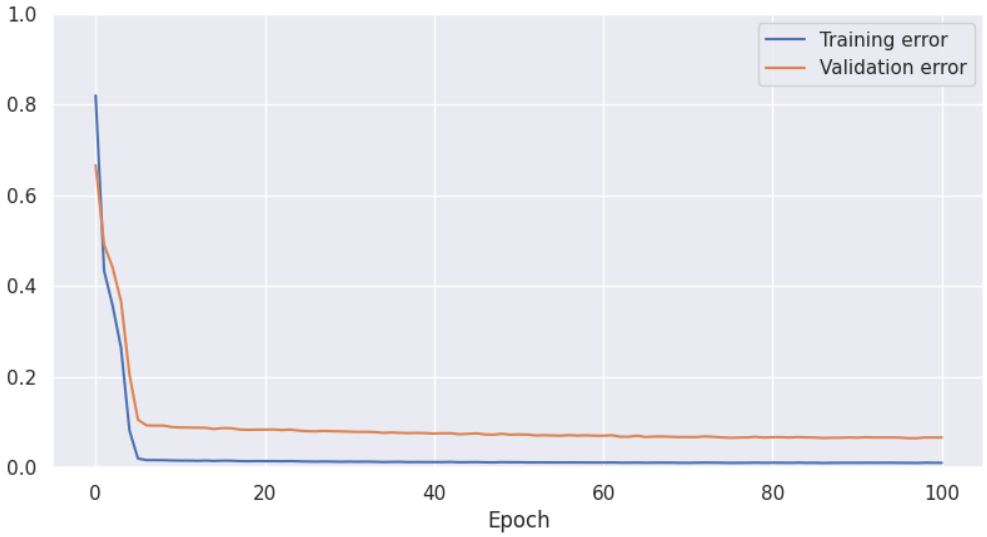
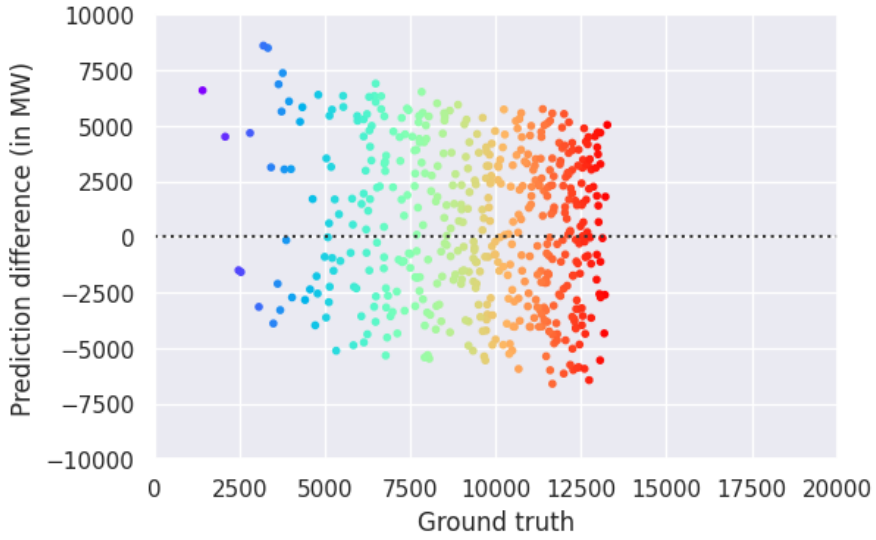


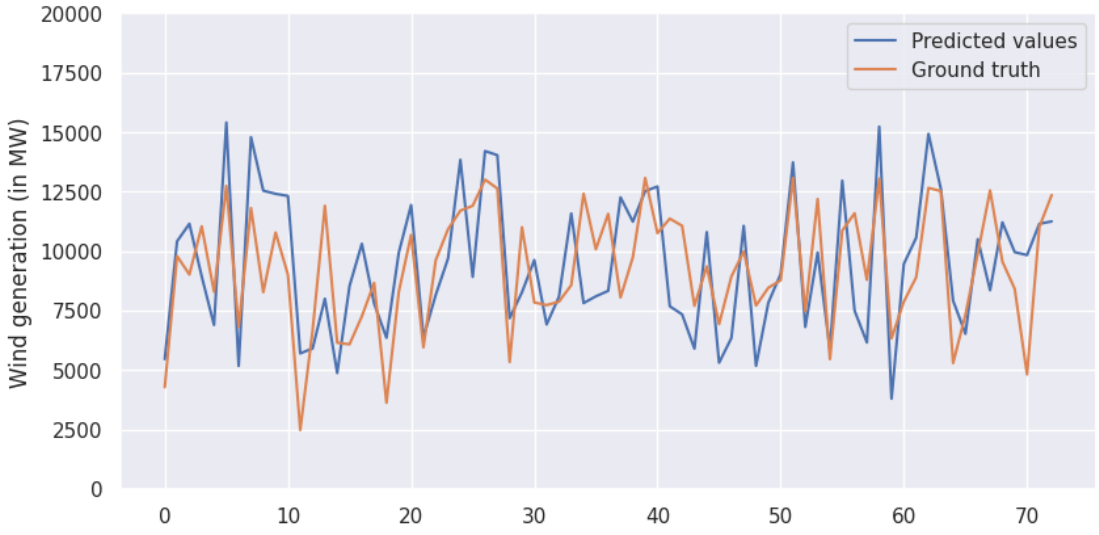
Figure 4.10: Training and validation loss (MSE) of the RNN-based model, using UK data with hourly resolution.

(a)

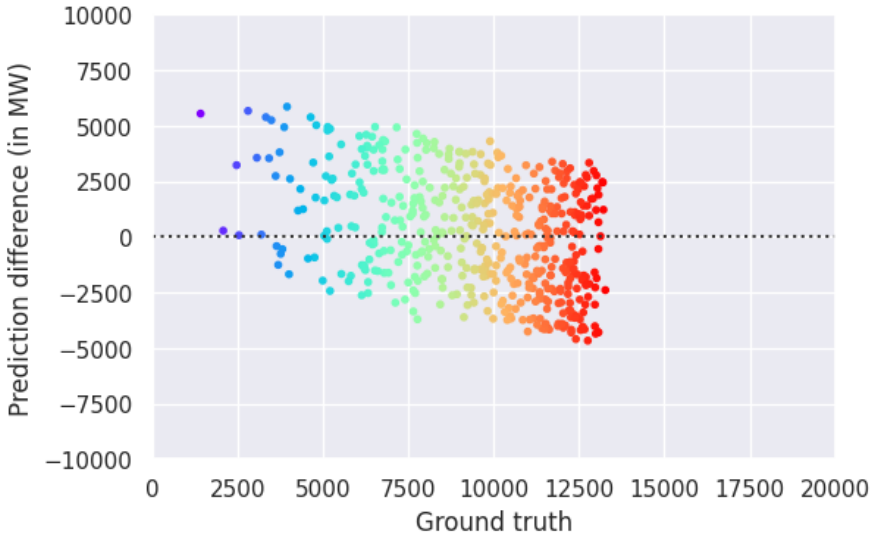


(b)

Figure 4.11: (a) FFNN prediction in actual wind generation compared to ground truth in a 72-hour stretch drawn from the test set, considering UK data with hourly resolution. (b) The resulting Bland-Altman plot considering all the test set.



(a)



(b)

Figure 4.12: (a) RNN prediction in actual wind generation compared to ground truth in a 72-hour stretch drawn from the test set, considering UK data with hourly resolution. (b) The resulting Bland-Altman plot considering all the test set.

4.1.4 Comparing yearly and monthly training

The monthly training, defined in Sec. 4.1.1, consists of training a neural network with all the same months in the training set and testing it with the same month of the test set. This means that 12 different predictors are to be trained, one per each month. In our case, each predictor (e.g., the one for January) is fed with data from 2016 and 2017 and is tested on the same month of the year 2018.

The main problem that arises in this scenario is overfitting, since the training samples are extremely limited. The weak point of this approach, indeed, is that 3 years of data actually correspond to 3 months of data, since every predictor only takes its "slice". Moreover, if the model overfits the training data, it is possible that the performance deteriorates as the epochs go by. In our experiment, we notice that the best performing model (for both FFNN and RNN) is around the 30th epoch, after which the prediction accuracy begins to decrease slowly.

	Yearly		**Monthly**	
Month	*FFNN*	*RNN*	*FFNN*	*RNN*
January	24.4	19.8	23.7	19.1
February	23.9	19.1	23.4	18.5
March	20.6	17.2	20.6	16.4
April	20.8	17.6	17.9	16.7
May	18.5	15.0	17.9	14.0
June	19.0	13.2	17.1	12.1
July	12.4	10.4	11.0	9.9
August	14.6	11.2	14.1	9.8
September	16.7	13.4	16.8	12.8
October	18.3	15.8	17.7	16.3
November	19.9	16.7	19.6	17.6
December	22.0	19.4	25.0	21.4
	18.9	**14.2**	**17.0**	**13.8**

Table 4.2: Mean Absolute Percentage Error (MAPE) with FFNN and RNN, using the yearly and monthly training strategy, with monthly granularity.

The results are shown in Tab. 4.2. We can see that the monthly training yields, on average, a slightly lower error in prediction. In particular, for both FFNN and RNN, the monthly training approach seems to be beneficial in the summer months, while in the winter this is not true, or at least it is not for all the months. Moreover, the FFNN-based predictor lowers the average MAPE from 18.9 to 17.0 (thus, a non-negligible margin). The same cannot be said for RNN, which only gains a

0.4% in average MAPE (from 14.2 down to 13.8). In the end, training a different model for each month seems to be only moderately beneficial, especially because it comes at a cost of training time, lack of generalization, and overfitting problems.

4.2 Hours-ahead forecast

Short-term wind power forecasting is an extremely important field of research for the energy sector, as the system operators must handle an important amount of fluctuating power from the increasing installed wind power capacity. The time scales concerning short-term prediction are in the order of some days for the forecast horizon and from minutes to hours for the time-step [26].

In this section, we explore the possibility of forecasting wind power generation at different prediction horizons in order to determine how much difference in prediction accuracy we can get. For obvious reasons, we expect the accuracy to decrease as the forecast horizon increases. For details about the difference approaches in hours-ahead forecast, see Sec. 3.6.

Hours-ahead	*1 FFNN-6 outputs*	*1 FFNN-1 output*	*6 FFNN-1 output*
t+1	18.9	18.9	18.9
t+2	36.0	35.9	34.2
t+3	39.5	38.6	36.4
t+4	45.8	42.2	39.0
t+5	47.7	44.3	39.9
t+6	49.4	45.6	40.4

Table 4.3: Mean Absolute Percentage Error (MAPE) on the DE dataset, with the different approaches at different time lags implemented using a FFNN.

Hours-ahead	*1 RNN-6 outputs*	*1 RNN-1 output*	*6 RNN-1 output*
t+1	14.2	14.2	14.2
t+2	22.7	22.4	22.4
t+3	28.8	27.8	27.5
t+4	34.2	33.5	32.8
t+5	38.8	36.1	35.0
t+6	39.4	37.4	35.9

Table 4.4: Mean Absolute Percentage Error (MAPE) on the DE dataset, with the different approaches at different time lags implemented using a RNN.

As we can see from Tab. 4.3 and Tab. 4.4, the last approach is the more suited for predicting wind power generation in a long prediction horizon. As a matter of fact, all methods perform very similar with a 2 or 3 hours-ahead forecast, as expected. In this case, the *6 NN-1 output* approach only has a slight edge over the other two approaches, while this gap extends rapidly when the prediction horizon goes up to 5-6 hours. Nonetheless, the accuracy decreases rapidly for both FFNN and RNN, meaning that the prediction becomes less and less reliable as the horizon becomes larger.

4.3 Estimating errors

In this section, we try to estimate errors and apply this estimation to predict how much the real value is going to differ from the predicted value. As stated in Sec. 1, in traditional power plants we need to guarantee the balance of demand and supply of energy and to ensure adequate power quality. If we are able to have some sort of feeling of what the prediction error is going to be, then we could potentially employ non-renewable energy sources to guarantee this equilibrium.

In this first step, we want to look at how the error is distributed across our base results achieved in Sec. 4.1. In order to do this, we use an unsupervised outlier detection method based on Support Vector Machines, called "OneClassSVM". The ν parameter of this algorithm is an upper bound on the fraction of margin errors and a lower bound of the fraction of support vectors relative to the total number of training examples. For example, we set it to 0.05, so we are guaranteed to find at most 5% of our training samples being misclassified (at the cost of a small margin) and at least 5% of our training samples being support vectors. In this case, our training samples correspond to the prediction errors.

As we can see in Table 4.5, a large fraction of the worst predictions made by the recurrent neural network is during the winter season, especially in January and February. During these months, the prediction is considerably worse than in the spring or in the summer. This is true for both DE and UK, although in the latter it is a little less evident, as data shows.

As for the error distribution per hour, we did not notice any pattern or correlation. Fig. 4.13 shows the error distribution per hour for the DE data. The daytime hours accounted for roughly the 52% of the total, while the nighttime hours accounted for the remaining 48%. Also conjugating this distribution over the single month or season did not result in any pattern or significant correlation. Thus, we can say that our prediction accuracy at a particular timestamp is not conditioned by the hour at which the prediction takes place.

% over total	*DE hourly*	*DE hourly (seasonal)*	*UK hourly*	*DE q-hourly*
January	16%	43%	12%	15%
February	15%		17%	15%
March	12%		10%	11%
April	6%	17%	8%	7%
May	7%		4%	6%
June	4%		8%	6%
July	4%	14%	2%	3%
August	7%		5%	5%
September	3%		6%	5%
October	8%	26%	9%	7%
November	9%		11%	10%
December	9%		9%	10%

Table 4.5: Prediction error distribution per month over the total prediction errors outliers. The predictions refer to the ones made in Sec. 4.1 using the RNN best performing model.

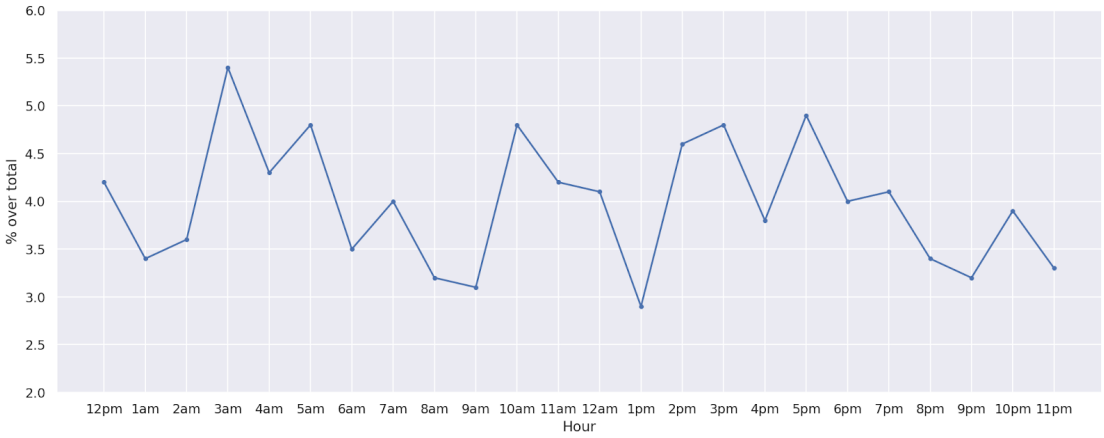
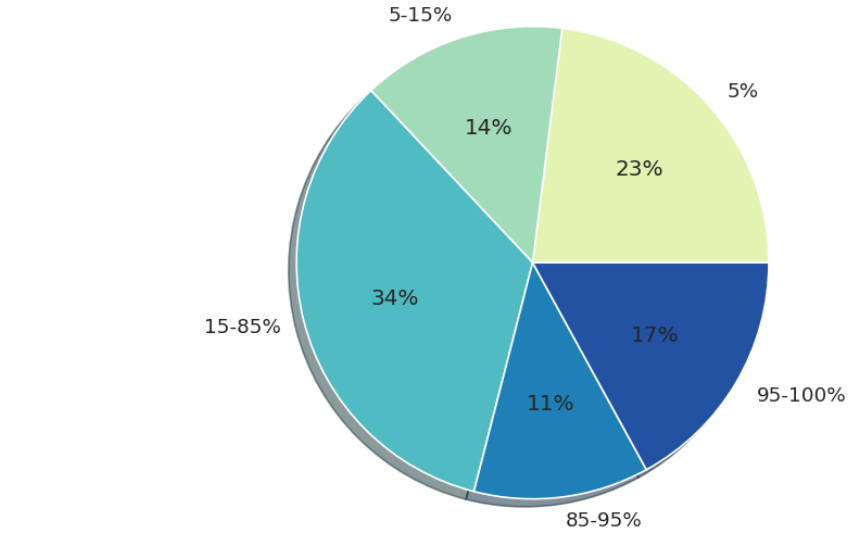
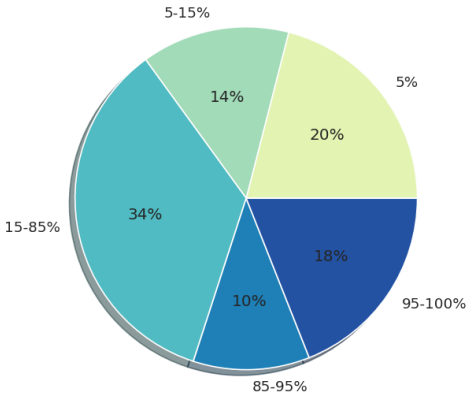


Figure 4.13: Prediction error distribution per hour over the total prediction errors outliers.

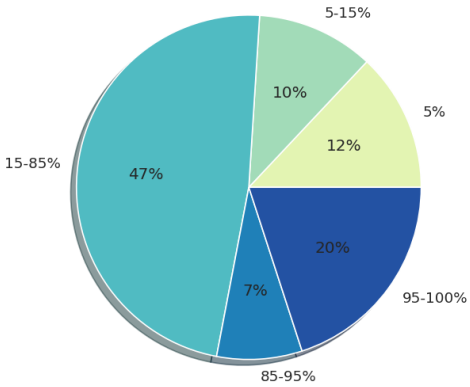
Next, we perform some percentile analysis to detect whether particularly high or particularly low target values of wind power generation could result in a loss of prediction accuracy. In particular, we want to know if our neural networks are able to keep the errors low when the ground truth is an anomaly or an outlier value. In order to do this, we look at the distribution of the prediction errors over the percentile values in the test set.



(a)



(b)



(c)

Figure 4.14: (a) Percentile analysis for the DE dataset taken with hourly resolution. (b) As for (a), but with quarter-hourly resolution. (c) As for (a) but using UK data.

Fig. 4.14a shows that the lowest 5% values in the entire test set (DE) account for 23% of the worst made predictions. The next 10% lowest values (i.e., the range 5-15%) also account for 14%. In a similar fashion, the predictions made for the highest 5% values (range 95-100%) constitute a non-negligible 17%. Likewise, a different time resolution (Fig. 4.14b) or different geolocations (Fig. 4.14c) achieved close results. This suffices to say that the prediction accuracy heavily decreases

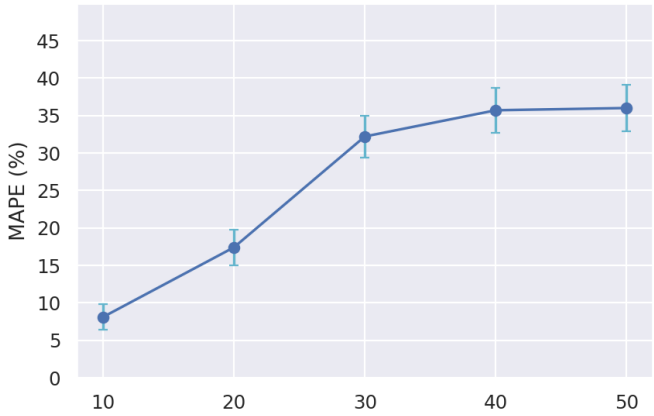
when we have to predict maxima or minima of power generation. This is expected as we employed several regularization techniques to prevent the model from overfitting on few extreme cases. We did not try to build two models — one on the "normal" data and one on outliers — as we are dealing with time-series data, neither we considered removing outliers from the dataset.

At this point, we know that anomalies in the target feature can cause our models to make bad predictions in the short-term time horizon. We investigate this aspect even further, suggesting that if the sample to be predicted — namely, y_{t+1} — differs more than a certain threshold from the samples immediately before, then the prediction can result in a loss of accuracy.

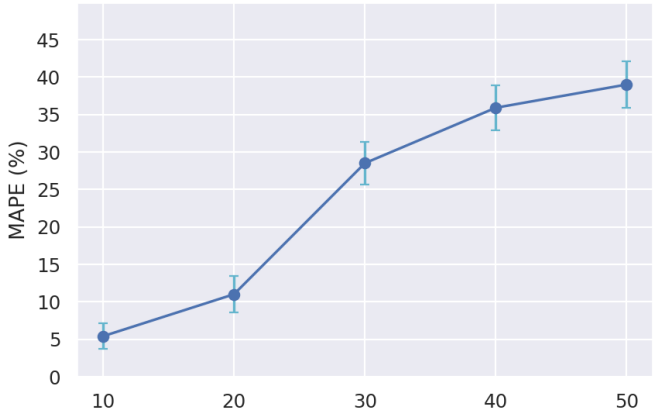
To verify this, we use the following algorithm. We name n the number of samples that we consider "immediately before" $(y_{t-n+1}, y_{t-n+2}, \ldots, y_t)$ and we compute the average of their values, that we call μ_t. Then, we take the upcoming value y_{t+1} and compute the percentage difference with respect to μ_t. For instance, if $\mu_t = 100MW$ and $y_{t+1} = 115MW$, the percentage difference is 15. Thus, we construct 5 different classes, where each class contains all the samples whose percentage difference is not greater than a certain threshold, but greater than the threshold immediately before (we use thresholds 10, 20, 30, 40, 50). In the above example, the sample would be put in the class with threshold 20, since 15 is greater than the first threshold 10, but less than 20.

Once this is all set, we compute the prediction error for all the entries in a class, and take their average value along with the standard deviation. Fig. 4.15a, Fig. 4.15b, and Fig. 4.15c show the average error and deviation per class using respectively $n = 3, n = 5, n = 7$ with the RNN architecture on the DE dataset. As we can see, the prediction error increases almost linearly as long as the percentage difference increases. For instance, if we take $n = 3$, we see that the average MAPE made in the prediction phase when the percentage difference is not greater than 10 is 7.8%, while it goes up to 35.7% when this difference is greater than 40 and not greater than 50. The same experiment was made on the UK data (Fig. 4.17), showing similar results.

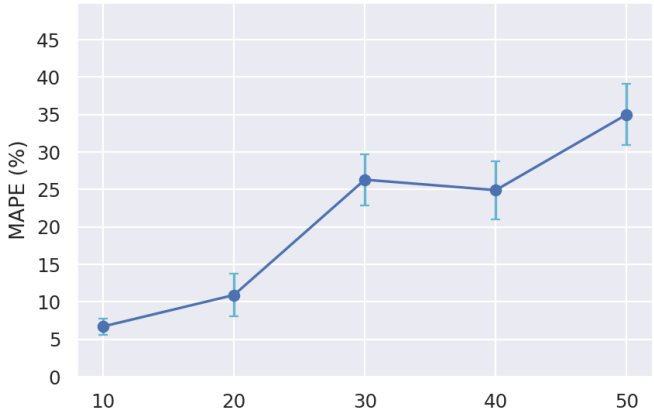
As a matter of fact, our models perform quite well when the data remains more or less constant over time. Nonetheless, the performance deteriorates rapidly when the generated wind power changes suddenly by a large margin. This behavior is consistent with the one we discussed in Sec. 4.1, where we noticed an overestimation of local maxima and an underestimation of local minima. Indeed, these peak values contribute to the classes with an high percentage difference, thus resulting in an increase of average error.



(a)

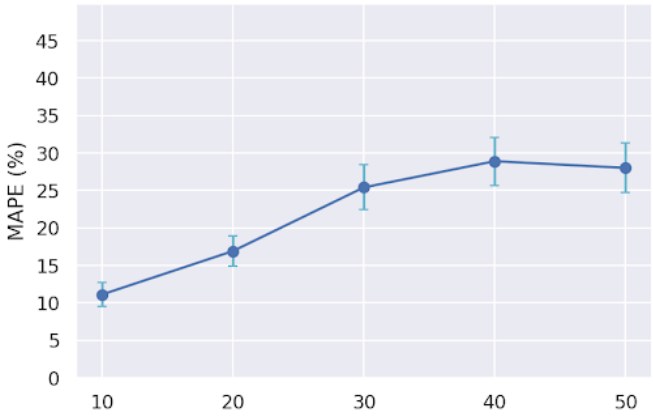


(b)

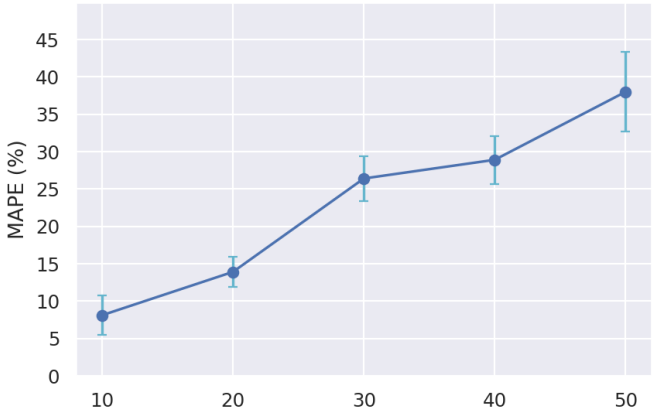


(c)

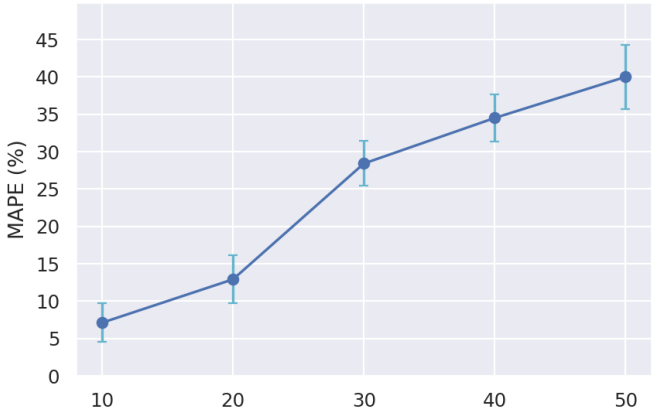
Figure 4.15: (a) Average prediction accuracy per class where each class indicator is the threshold percentage difference for the upcoming sample with respect to the average of the $n = 3$ preceding samples (DE). (b) As for (a), but with $n = 5$. (c) As for (a), but with $n = 7$.



(a)



(b)



(c)

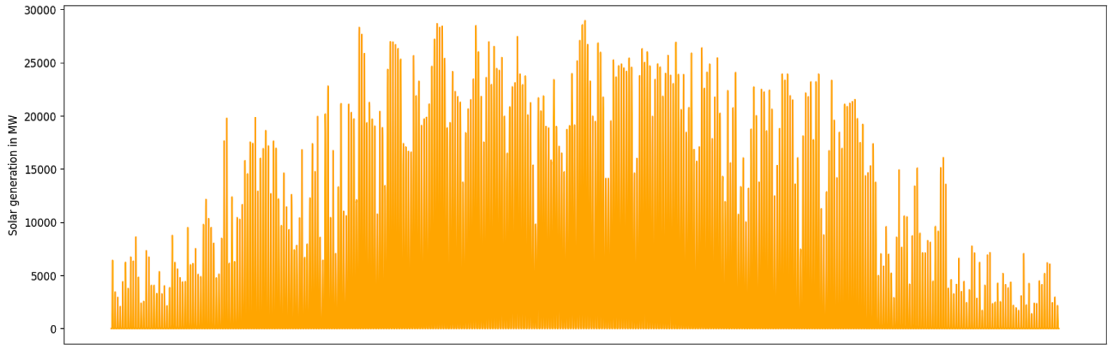
Figure 4.16: (a) Average prediction accuracy per class where each class indicator is the threshold percentage difference for the upcoming sample with respect to the average of the $n = 3$ preceding samples (UK). (b) As for (a), but with $n = 5$. (c) As for (a), but with $n = 7$.

4.4 Hybrid scenario: solar and wind

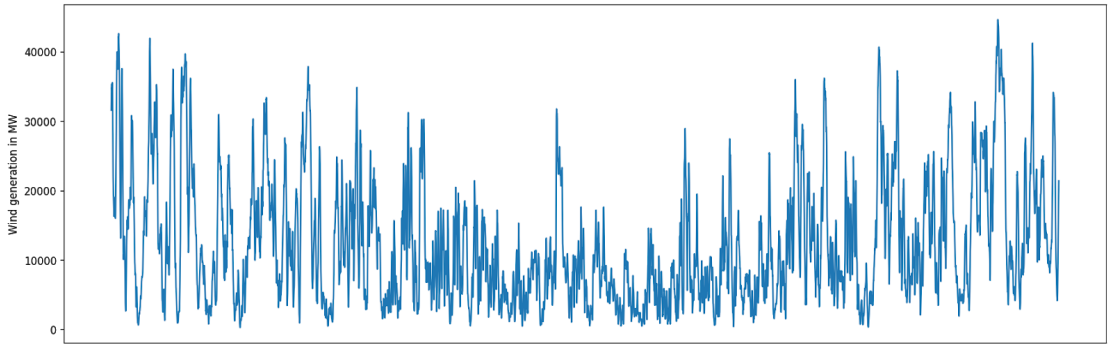
So far we have discussed solely about wind power generation and how its prediction can be affected by a variety of factors. We also underlined some important characteristics of wind power, including its extreme variability, discontinuity, and the presence of a clear correlation with seasons. In this section, we explore the possibility of including solar energy in the context of a power grid, and forecasting it alongside wind power.

The main advantage of using solar energy is its extreme predictability. During the day, in fact, the generated solar power usually follows a precise curve, while reaching and keeping a value of 0 during the night. The production of solar energy in the Northern Hemisphere is naturally higher in the summer than in the rest of the months, as we can see in Fig. 4.17a. From this graph, which plots the total solar power generation in Germany in 2018, it is evident that, aside night hours, the solar track is almost continuous. This is especially true when we compare it to the total wind power generation in the same period (Fig. 4.17b). The wind graph exposes its maxima in the winter (extreme left and extreme right), while keeping blatantly low values in the summer (center). Also, in this case we have no clear distinction between the day and the night (see also Fig. 4.13).

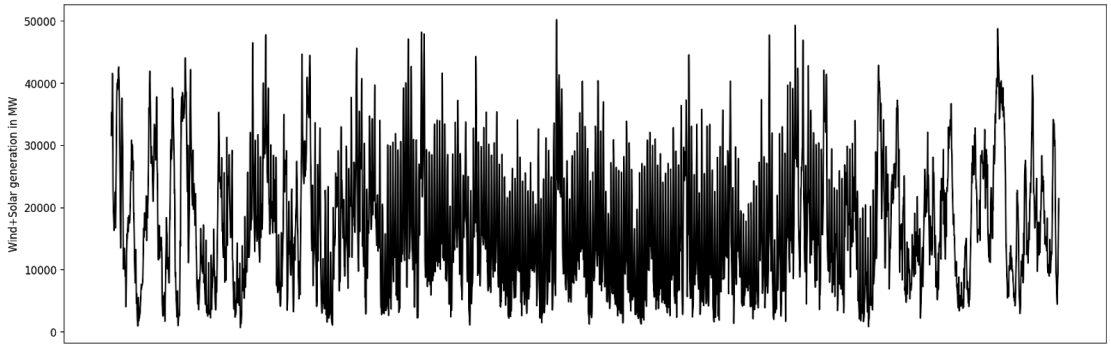
The total solar and wind power generation is shown in Fig. 4.17c. The first thing to notice is that, despite the bigger "density" in the summer, the minimum and maximum value of each month (i.e., if we ideally divide the graph in 12 equally wide sections) is very similar. Indeed, in the winter months the wind power generation is predominant and actually reaches its peak value, while the solar energy production is extremely limited. Diversely, the solar energy is prevalent in the summer. This also explains the "density", which reflects the intermittent nature of the solar. This translates in a sort of "balance effect", in which the transition between different months and seasons is almost seamless, when we consider their aggregate. Nevertheless, it is worth to notice that the cumulative wind power exceeds the cumulative solar power in each month, with the exception of July (Fig. 4.19).



(a)



(b)



(c)

Figure 4.17: (a) Solar power generation (MW) in Germany in 2018. (b) Wind power generation (MW) in Germany in 2018. (c) Total solar and wind power generation (MW) in Germany in 2018.

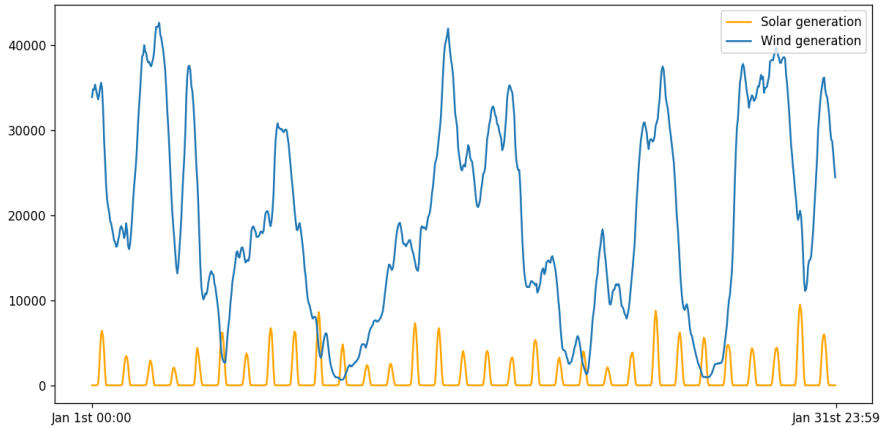
Fig. 4.18 exhibits the solar and wind tracks for each month of 2018. In January and December, for instance, we can see that the wind generation floats between peak values of over 40GW to rare moments when the production goes below 5GW. In the same time period, the solar wind generation never exceeds 10GW, while also maintaining very long periods of 0s for the entire duration of the nights, which are longer than in the summer (since in the Northern Hemisphere nighttime is longer in the winter).

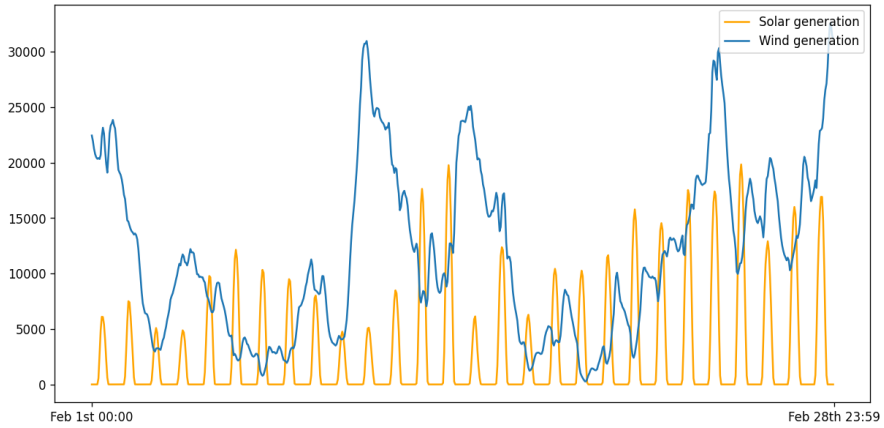
Things start to balance out in April and May, where we can often see the solar energy exceed the wind energy. During these months, the wind power generation exceeds 30GW only twice, while solar generated power exceeds 20GW in the majority of daytime. Also, the interval between between the "parabolas" of daytime becomes tighter and tighter as nighttime duration decreases. From June to August, we can notice that wind energy decreases and is overstepped by solar energy in almost the totality of daytime hours. From the end of October until December, we gradually go back to the same situation we experience in January.

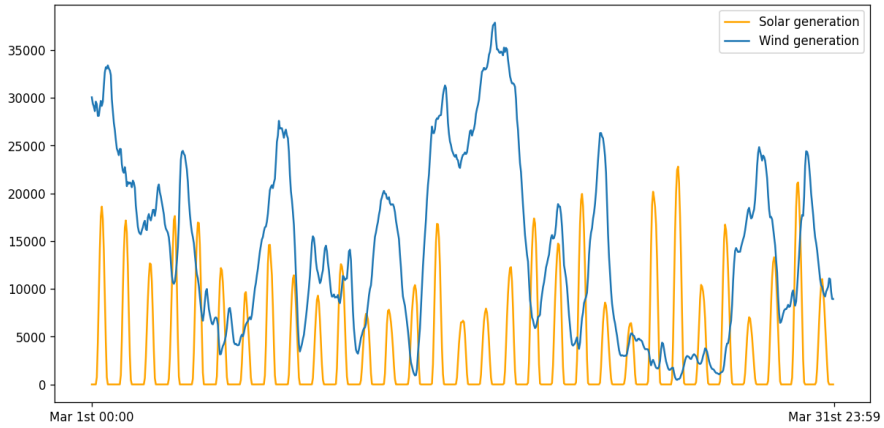
The prediction of the total solar and wind power, in the context of a smart grid, is crucial to keep the balance between demand and supply. At any point, we would like to know the exact amount of generated power, which can be eventually stored for later use if demand does not keep up with it. In fact, unlike traditional coal or natural gas-fired power plants, the output of solar panels and wind turbines cannot be quickly increased to meet demand.

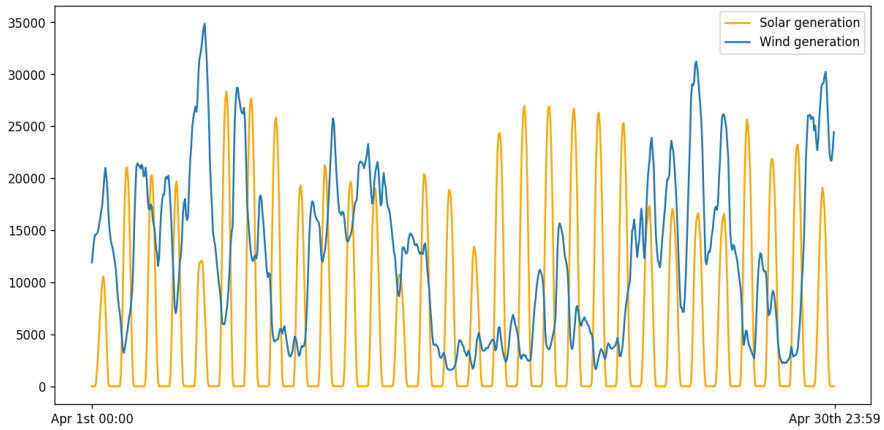
In this context, we cluster data having in mind 5 different scenarios:

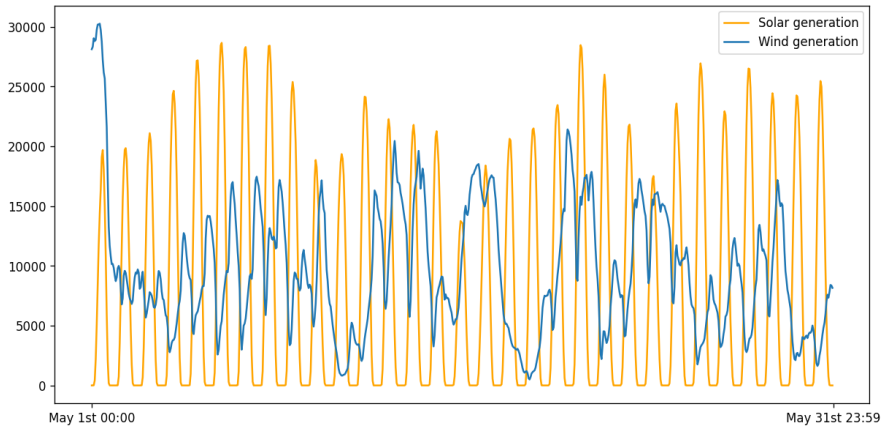
- Cluster 1: extremely high production, almost certainly to be partly stored;
- Cluster 2: high production, likely to be partly stored;
- Cluster 3: medium production, can be partly stored or drawn from storage;
- Cluster 4: low production, likely to partly draw from storage;
- Cluster 5: extremely low production, almost certainly to partly draw from storage.

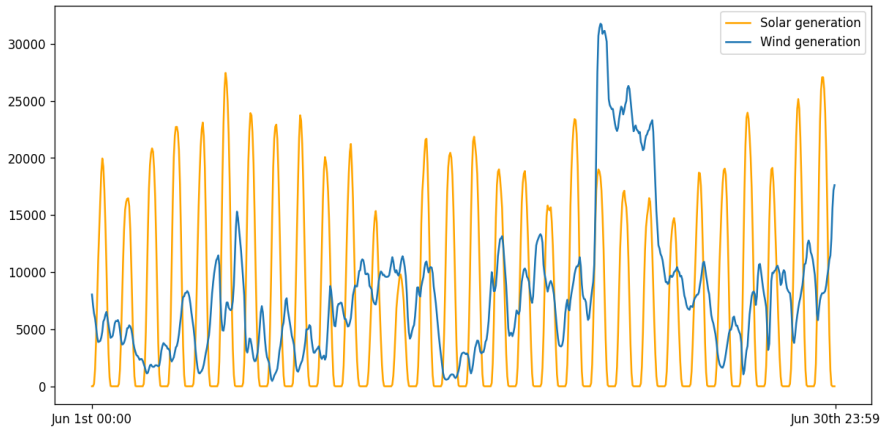

Solar generation
Wind generation
40000
30000
20000
10000
0
Jan 1st 00:00
Jan 31st 23:59

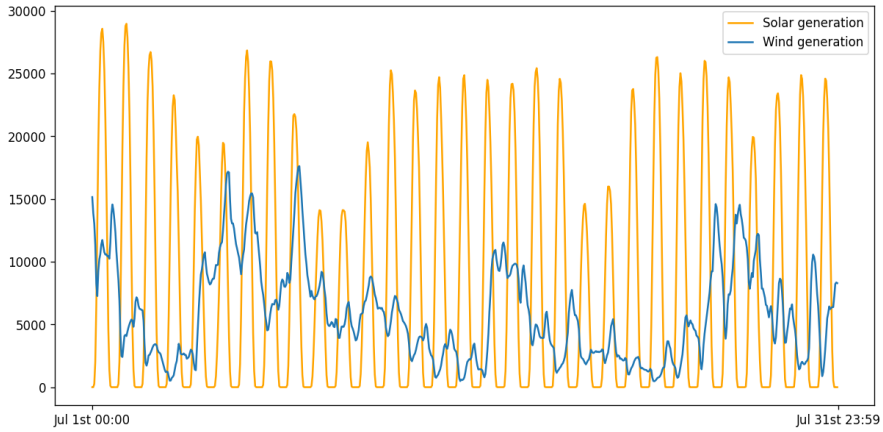

Solar generation
Wind generation
30000
25000
20000
15000
10000
5000
0
Feb 1st 00:00
Feb 28th 23:59

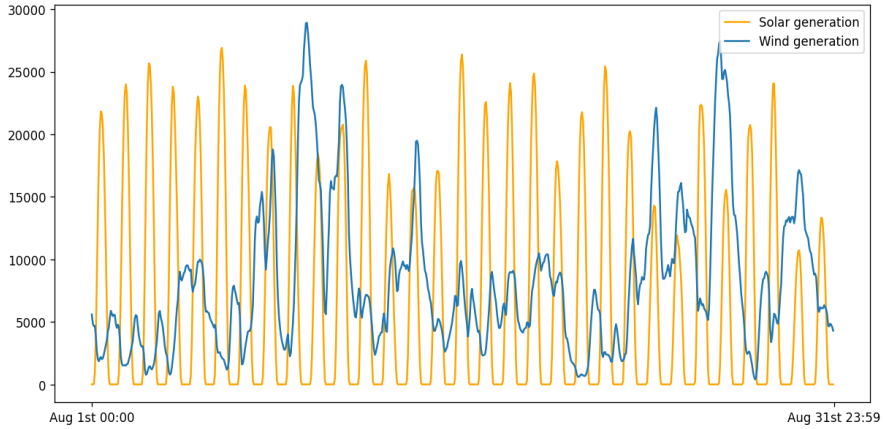

Solar generation
Wind generation
35000
30000
25000
20000
15000
10000
5000
0
Mar 1st 00:00
Mar 31st 23:59

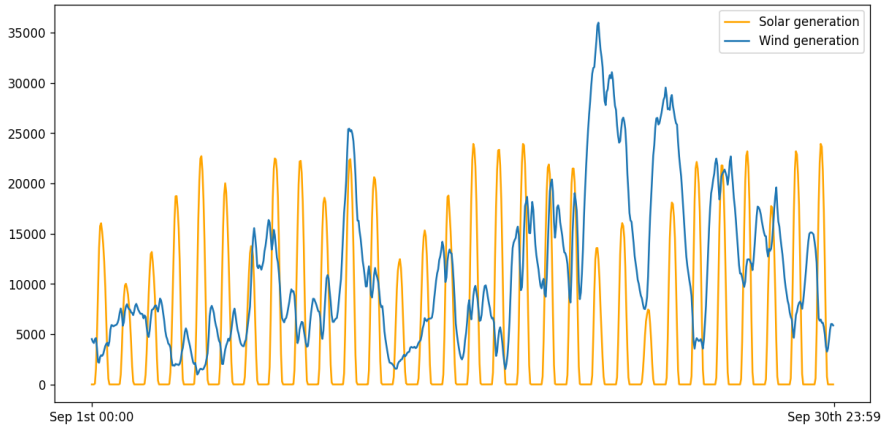

Solar generation
Wind generation
35000
30000
25000
20000
15000
10000
5000
0
Apr 1st 00:00
Apr 30th 23:59


Solar generation
Wind generation
30000
25000
20000
15000
10000
5000
0
May 1st 00:00
May 31st 23:59


Solar generation
Wind generation
30000
25000
20000
15000
10000
5000
0
Jun 1st 00:00
Jun 30th 23:59


Solar generation
Wind generation
30000
25000
20000
15000
10000
5000
0
Jul 1st 00:00
Jul 31st 23:59


Solar generation
Wind generation
30000
25000
20000
15000
10000
5000
0
Aug 1st 00:00
Aug 31st 23:59


Solar generation
Wind generation
35000
30000
25000
20000
15000
10000
5000
0
Sep 1st 00:00
Sep 30th 23:59

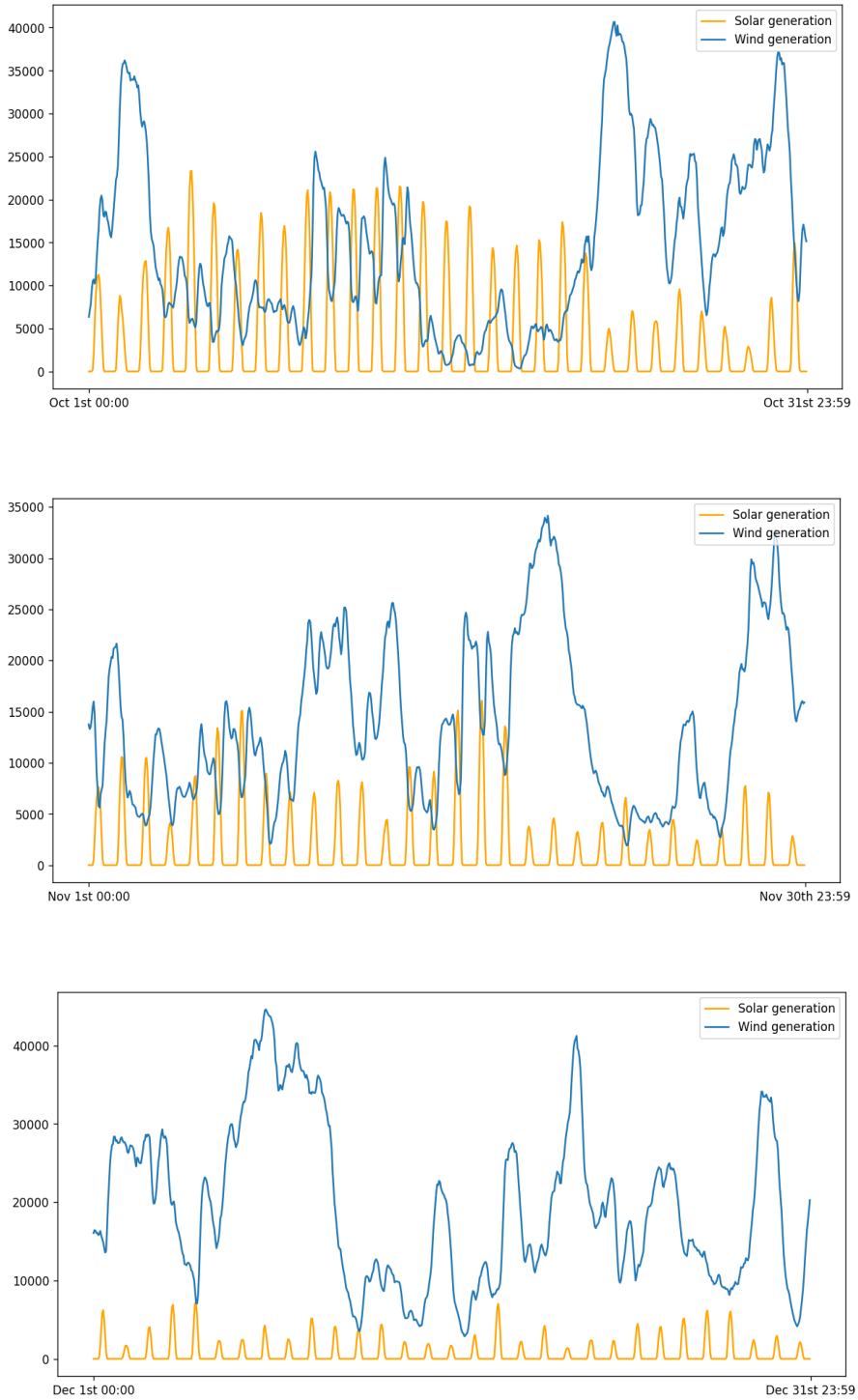


Figure 4.18: Solar and wind power generation per month in Germany in 2018.

The clustering of the data enables us to differentiate prediction accuracy in each of the aforementioned scenarios. Fig. 4.19 depicts the distribution how the data points are distributed among the 5 clusters in each month. As we said before, Cluster 1 and 5 data points are present in each month, with December being the one with the most Cluster 1 points and June being the one with the most Cluster 5 points.

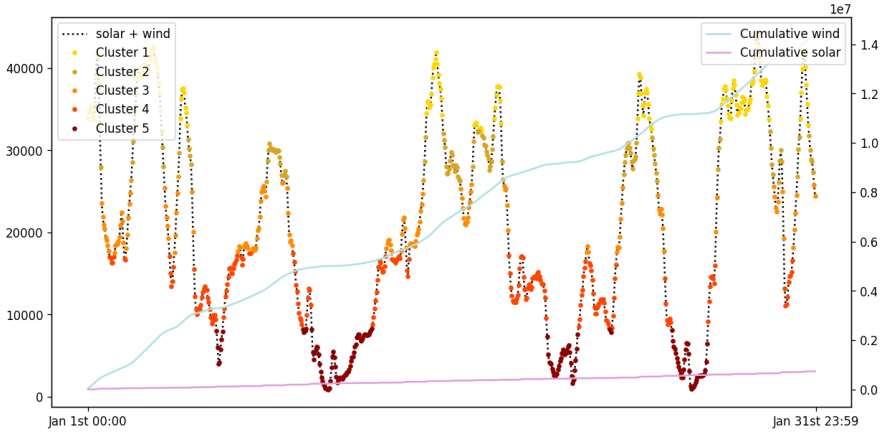
As far as the prediction is concerned, we use the same methodology as in Sec. 4.1. The features concerning solar energy are already included in the original dataset (see Tab. 2.1). The only difference with respect to wind-only prediction is the number of PCA components to reach a minimum explained variance of 90%, since we are dealing with a larger number of features. In this case, 8 components are chosen, hence to explain 92% of the variance.

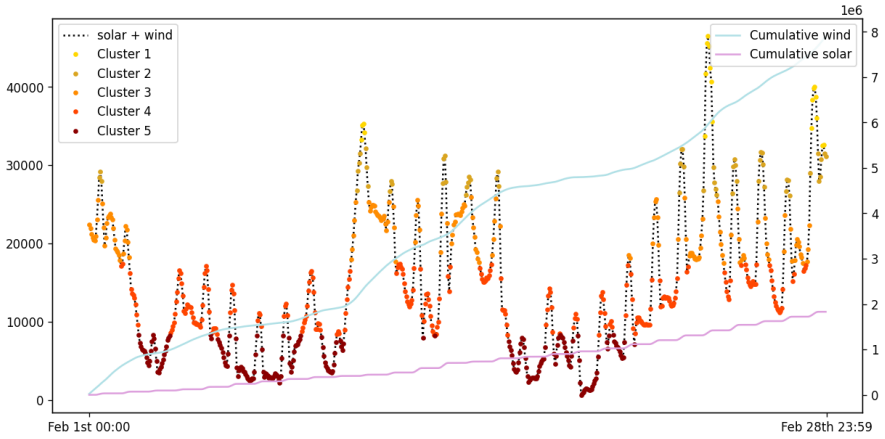
Again, the FFNN and RNN-based models are chosen as our default, along with the same hyperparameters. Tab. 4.6 summarizes the results of the prediction. We can notice that Cluster 1 and Cluster 2 data points are by far the most difficult to predict for our models. Indeed, a lot of these points belonging to these clusters are centered around winter months, where the production of wind energy (the hardest to predict) outsteps the production of solar energy (the easiest to predict).

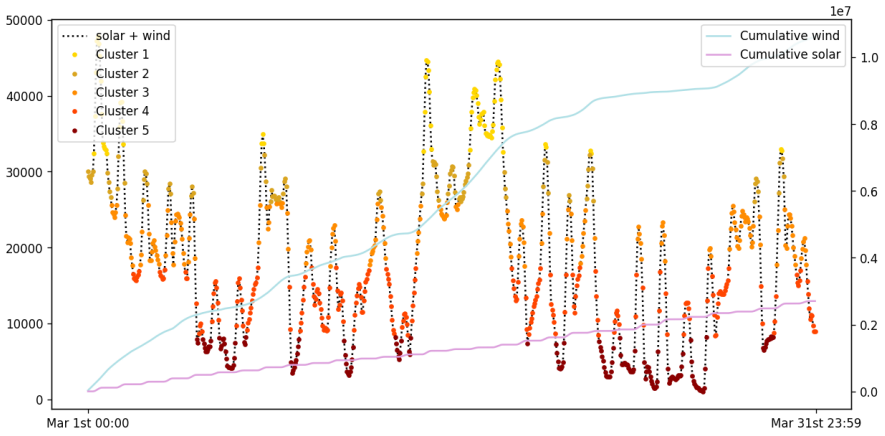
Overall, the average MAPE is considerably lower than the one we got in a wind-only prediction in Sec. 4.1, thus we can say that the solar track in a solar and wind scenario "softens" the difficulty of predicting only wind power.

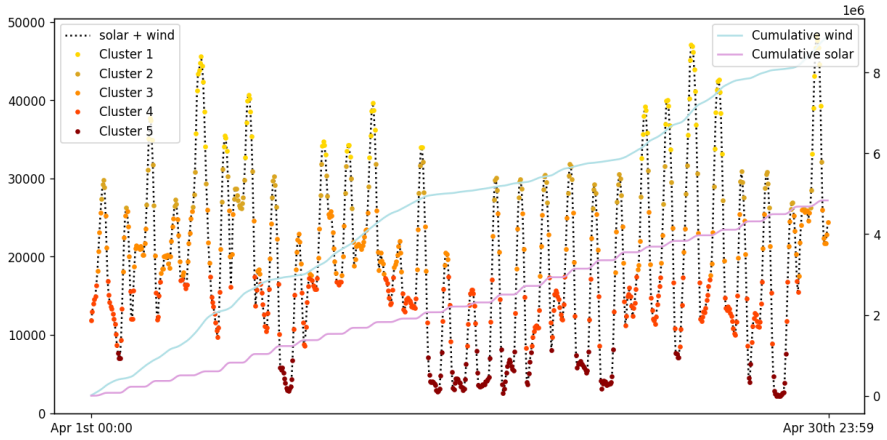
	FFNN		**RNN**	
Cluster	*MAE*	*MAPE*	*MAE*	*MAPE*
Cluster 1	9861.4	22.8	9067.0	21.5
Cluster 2	8914.7	20.3	8481.2	19.7
Cluster 3	3994.8	8.4	3338.5	7.8
Cluster 4	4765.9	11.3	4490.6	10.8
Cluster 5	7688.5	14.5	7239.4	14.0
	6048.3	**14.9**	**5110.8**	**11.2**

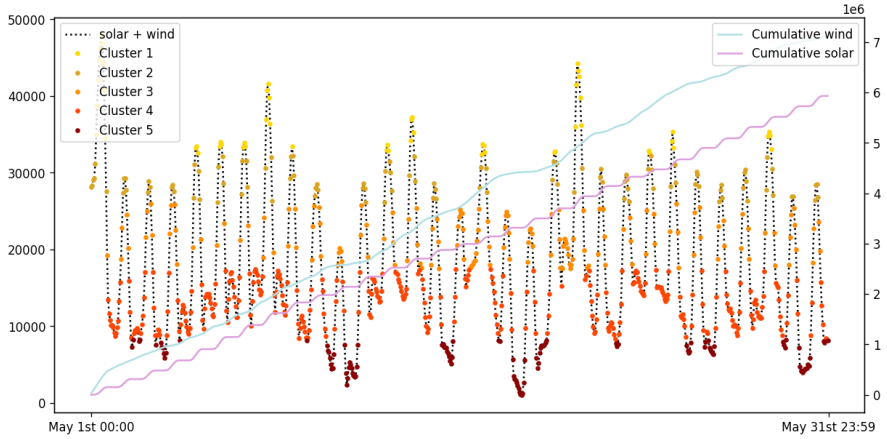
Table 4.6: Mean Absolute Error (MAE) and Mean Absolute Percentage Error (MAPE) with FFNN and RNN per cluster, while predicting total solar and wind power generation.

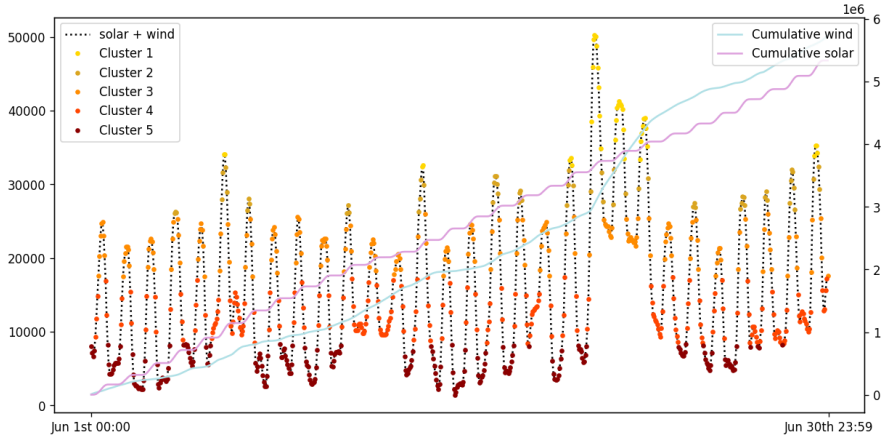

1e7
solar + wind
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Cumulative wind
Cumulative solar
40000
30000
20000
10000
0
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Jan 1st 00:00
Jan 31st 23:59

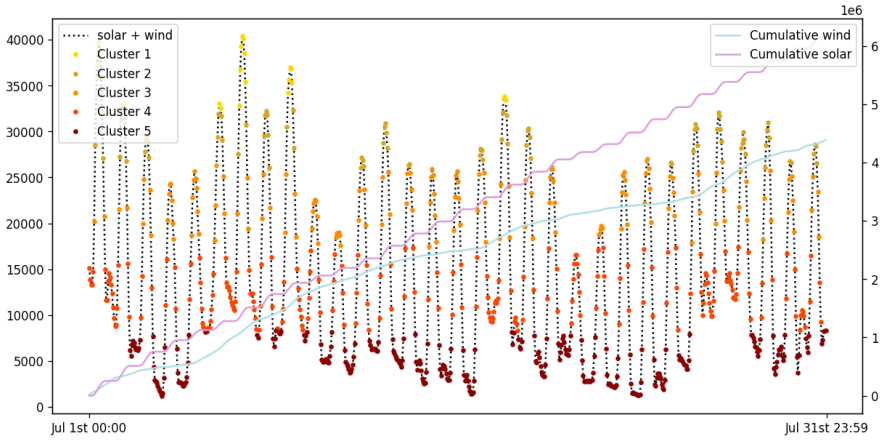

1e6
solar + wind
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Cumulative wind
Cumulative solar
40000
30000
20000
10000
0
8
7
6
5
4
3
2
1
0
Feb 1st 00:00
Feb 28th 23:59

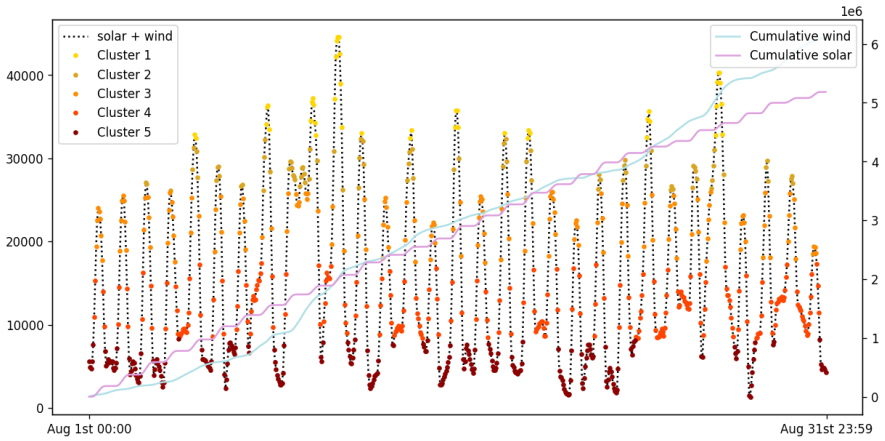

1e7
solar + wind
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Cumulative wind
Cumulative solar
50000
40000
30000
20000
10000
0
1.0
0.8
0.6
0.4
0.2
0.0
Mar 1st 00:00
Mar 31st 23:59

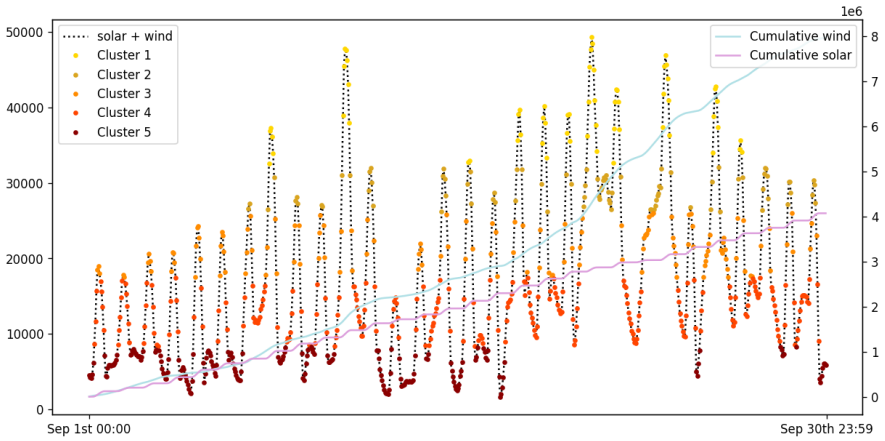

1e6
solar + wind
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Cumulative wind
Cumulative solar
50000
40000
30000
20000
10000
0
8
6
4
2
0
Apr 1st 00:00
Apr 30th 23:59


1e6
solar + wind
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Cumulative wind
Cumulative solar
50000
40000
30000
20000
10000
0
7
6
5
4
3
2
1
0
May 1st 00:00
May 31st 23:59


1e6
solar + wind
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Cumulative wind
Cumulative solar
50000
40000
30000
20000
10000
0
6
5
4
3
2
1
0
Jun 1st 00:00
Jun 30th 23:59


1e6
solar + wind
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Cumulative wind
Cumulative solar
40000
35000
30000
25000
20000
15000
10000
5000
0
6
5
4
3
2
1
0
Jul 1st 00:00
Jul 31st 23:59


1e6
solar + wind
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Cumulative wind
Cumulative solar
40000
30000
20000
10000
0
6
5
4
3
2
1
0
Aug 1st 00:00
Aug 31st 23:59


1e6
solar + wind
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Cumulative wind
Cumulative solar
50000
40000
30000
20000
10000
0
8
7
6
5
4
3
2
1
0
Sep 1st 00:00
Sep 30th 23:59

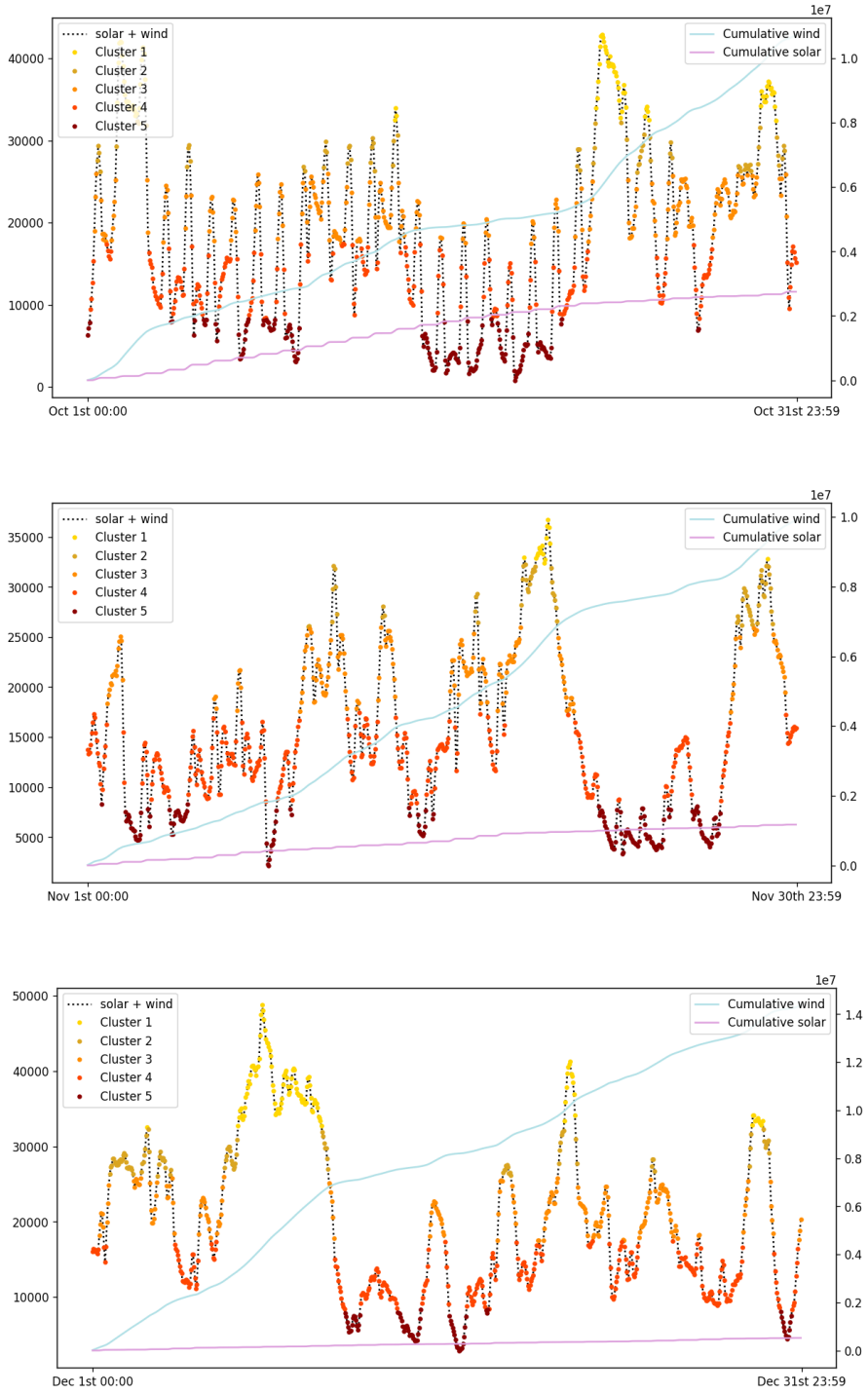


Figure 4.19: Clustered data points and cumulative solar and wind power generation in Germany in 2018.

Chapter 5

Conclusions

The objective of the thesis was to propose a machine learning framework to develop an accurate prediction system in the context of renewable energy sources. Machine learning, indeed, has become a fundamental aspect of the decision process in both the generation and the distribution of energy. For instance, in the context of smart grids, machine learning offers opportunities to progress by providing many facilities that are not limited to energy supply and demand balance, but to ensure providing the quality criteria of energy and energy measurement.

Wind energy was selected as our main focus, because of its high availability and predictability. The OPSD dataset provided us a great starting point for characterizing European power plant fleets and their associated transmission network. Their time-series offered us a great insight on data aggregated by country, control areas, and bidding zones. Also, the data being available in different time granularities enabled us to experiment with prediction time lags and horizons. After having chosen German data as our starting point, we processed the data before the effective training phase. In particular, we normalized the data in order to reduce the high internal variance and we applied PCA to lower the dimensionality of the dataset, thus allowing ourselves to deal with simpler and dense data.

We subsequently introduced the concept of neural network and deep learning, focusing on two main architectures: the feedforward (FFNN) and recurrent (RNN). While FFNN is generally considered the vanilla approach to machine learning, recurrent networks — whose outputs are fed back into the network, unlike FFNN — are usually preferred when it comes to time-series, since their "memory" mechanism can significantly boost up their performance in prediction.

The results clearly showed an attunement with the literature, with RNN-based models outperforming FFNN-based ones in all the experiments we conducted. The

main problem we faced in this segment was the tendency of the models to overfit the data and to generalize poorly on new inputs. Also, we established that higher resolution data can help neural networks in better understanding patterns, since using quarter-hourly data in place of hourly data resulted in an increase of accuracy. We then wanted to test how the same architectural models generalize on data from different countries and with other characteristics. For this purpose, we repeated the same experiment on aggregated data from the United Kingdom which gave us a positive result.

With these results in mind, we increased the prediction horizon up to 6 hours, establishing a correlation between the prediction window and the overall accuracy of the models. Further analysis was also conducted to try and estimate the errors of the neural networks beforehand. The outcome of this examination was that our models have their performance deteriorated when anomaly values have to be predicted, or when consecutive samples are strongly discontinuous. Last but not least, we probed into a hybrid scenario including both wind and solar power, in which we noticed that our models' performance improved when the solar power exceeded wind power, especially in the summer.

References

[1] Thomas Love and Cindy Isenhour. «Energy and economy: Recognizing high-energy modernity as a historical period». In: *Economic Anthropology* 3.1 (), pp. 6–16. DOI: https://doi.org/10.1002/sea2.12040 (cit. on p. 1).

[2] Phillipp Beiter, Michael Elchinger, and Tian Tian. *2016 renewable energy data book*. Tech. rep. National Renewable Energy Lab.(NREL), Golden, CO (United States), 2017 (cit. on p. 1).

[3] Kasun S Perera, Zeyar Aung, and Wei Lee Woon. «Machine learning techniques for supporting renewable energy generation and integration: a survey». In: *International Workshop on Data Analytics for Renewable Energy Integration*. Springer. 2014, pp. 81–96 (cit. on p. 3).

[4] G. Vallero, D. Renga, M. Meo, and M. A. Marsan. «Greener RAN Operation Through Machine Learning». In: *IEEE Transactions on Network and Service Management* 16.3 (2019), pp. 896–908. DOI: 10.1109/TNSM.2019.2923881 (cit. on p. 4).

[5] Meysam Masoudi et al. «Green Mobile Networks for 5G and Beyond». In: *IEEE Access* 7 (Aug. 2019), pp. 1–1. DOI: 10.1109/ACCESS.2019.2932777 (cit. on p. 4).

[6] Bingqian Xu, Pengcheng Zhu, Jiamin Li, Dongming Wang, and You Xiaohu. «Joint Long-term Energy Efficiency Optimization in C-RAN with Hybrid Energy Supply». In: *IEEE Transactions on Vehicular Technology* PP (July 2020), pp. 1–1. DOI: 10.1109/TVT.2020.3007825 (cit. on p. 5).

[7] Inci Okumus and Ali Dinler. «Current status of wind energy forecasting and a hybrid method for hourly predictions». In: *Energy Conversion and Management* 123 (2016), pp. 362–371. ISSN: 0196-8904. DOI: https://doi.org/10.1016/j.enconman.2016.06.053. URL: http://www.sciencedirect.com/science/article/pii/S0196890416305428 (cit. on p. 6).

[8] Umit Cali and Vinayak Sharma. «Short-term wind power forecasting using long-short term memory based recurrent neural network model and variable selection». In: *International Journal of Smart Grid and Clean Energy* (Jan. 2019), pp. 103–110. DOI: 10.12720/sgce.8.2.103-110 (cit. on p. 6).

[9] Peiyuan Chen, Troels Pedersen, Birgitte Bak-Jensen, and Z. Chen. «ARIMA-Based Time Series Model of Stochastic Wind Power Generation». In: *Power Systems, IEEE Transactions on* 25 (June 2010), pp. 667–676. DOI: 10.1109/TPWRS.2009.2033277 (cit. on p. 7).

[10] Thanasis Sfetsos. «A comparison of various forecasting techniques applied to mean hourly wind speed time series». In: *Renewable Energy* 21 (Sept. 2000), pp. 23–35. DOI: 10.1016/S0960-1481(99)00125-1 (cit. on p. 7).

[11] M. Ding, L.-J Zhang, and Y.-C Wu. «Wind speed forecast model for wind farms based on time series analysis». In: *Electric Power Automation Equipment* 25 (Jan. 2005), pp. 32–34 (cit. on p. 7).

[12] S. Li. «Wind power prediction using recurrent multilayer perceptron neural networks». In: *2003 IEEE Power Engineering Society General Meeting (IEEE Cat. No.03CH37491)*. Vol. 4. 2003, 2325–2330 Vol. 4. DOI: 10.1109/PES.2003.1270992 (cit. on p. 7).

[13] «Open Power System Data. 2020.» In: *Data Package Time series*. DOI: https://doi.org/10.25832/time_series/2020-10-06 (cit. on p. 8).

[14] Svante Wold, Kim Esbensen, and Paul Geladi. «Principal component analysis». In: *Chemometrics and intelligent laboratory systems* 2.1-3 (1987), pp. 37–52 (cit. on p. 12).

[15] Mengyi Gong, Claire Miller, and Marian Scott. «Functional PCA for Remotely Sensed Lake Surface Water Temperature Data». In: *Procedia Environmental Sciences* 26 (2015). Spatial Statistics conference 2015, pp. 127–130. ISSN: 1878-0296. DOI: https://doi.org/10.1016/j.proenv.2015.05.015 (cit. on p. 15).

[16] Frank Rosenblatt. «The perceptron: a probabilistic model for information storage and organization in the brain.» In: *Psychological review* 65.6 (1958), p. 386 (cit. on p. 18).

[17] Paul J Werbos. «Backpropagation through time: what it does and how to do it». In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560 (cit. on p. 22).

[18] Sepp Hochreiter and Jürgen Schmidhuber. «Long short-term memory». In: *Neural computation* 9.8 (1997), pp. 1735–1780 (cit. on p. 26).

[19] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. «Learning long-term dependencies with gradient descent is difficult». In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166 (cit. on p. 26).

[20] Sepp Hochreiter and Jürgen Schmidhuber. «LSTM can solve hard long time lag problems». In: *Advances in neural information processing systems* (1997), pp. 473–479 (cit. on p. 26).

[21] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG] (cit. on p. 29).

[22] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG] (cit. on p. 32).

[23] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. «Dropout: A Simple Way to Prevent Neural Networks from Overfitting». In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html (cit. on p. 33).

[24] Greta Vallero, Daniela Renga, Michela Meo, and Marco Ajmone Marsan. «Processing ANN Traffic Predictions for RAN Energy Efficiency». In: *Proceedings of the 23rd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*. MSWiM '20. Alicante, Spain: Association for Computing Machinery, 2020, pp. 235–244. ISBN: 9781450381178. DOI: 10.1145/3416010.3423222. URL: https://doi.org/10.1145/3416010.3423222 (cit. on p. 35).

[25] Juan Yan, Xiaodong Zhao, and Kang Li. «On temporal resolution selection in time series wind power forecasting». In: *2016 UKACC 11th International Conference on Control (CONTROL)*. IEEE, pp. 1–6 (cit. on p. 43).

[26] João Paulo da Silva Catalão, Hugo Miguel Inácio Pousinho, and Vctor Manuel Fernandes Mendes. «An artificial neural network approach for short-term wind power forecasting in Portugal». In: *2009 15th International Conference on Intelligent System Applications to Power Systems*. IEEE. 2009, pp. 1–5 (cit. on p. 52).