CARL VON OSSIETZKY universität OLDENBURG

Deep Learning of Virtual Marine Sensors

A DISSERTATION PRESENTED BY
STEFAN OEHMCKE
TO
THE DEPARTMENT OF COMPUTING SCIENCE

FOR THE DEGREE OF
DR. RER. NAT.

CARL OF OSSIETZKY UNIVERSITY
OLDENBURG, LOWER SAXONY, GERMANY

REVIEWERS:
PROF. DR. OLIVER KRAMER,
PROF. DR. OLIVER ZIELINSKI,
PROF. DR. KORBINIAN RIEDHAMMER

DATE OF DISPUTATION:
20.04.2018

Stefan Oehmcke

Deep Learning of Virtual Marine Sensors

Abstract Understanding our marine ecosystems is important to protect the environment and promote the sustainable use of resources. Coastal processes are of of particular interest as the coastal regions have the largest maritime traffic and marine life. Observing these processes is a difficult task due to the harsh environmental conditions and the size of the area to be covered. To solve this task, arrays of robust sensors are required that can handle storms, saltwater, and the daily wear caused by the tides. The resulting amount of data is not manually processable and the chance that a sensor fails increases with growing infrastructure.

This thesis proposes an approach to build virtual sensors based on machine learning to replace broken physical sensors. These virtual sensors are trained with sensor data from the *Time Series Station Spiekeroog* (TSS) and the *Biodiversity-Ecosystem Functioning across marine and terrestrial ecosystems* (BEFmate) project in the Wadden Sea (German Bight). In the first part of my work, I begin by explaining the data and its preprocessing. Next, an unsupervised extreme event detection task on the TSS data with a subsequent expert evaluation is presented. Then I propose an imputation method for longer consecutively missing values as they appear in our datasets. The method utilizes linear interpolation as its first step, but penalizes these interpolated values based on the length of the gaps with a k-nearest neighbors approach (penalized DTWkNN ensemble). In the second part, I design a neural network architecture to model broken sensors from the BEFmate project. The foundation is a bidirectional recurrent neural network with long short-term memory (bLSTM) that utilizes my time dimensionality reduction method exponential piecewise approximate aggregation (exPAA). Then, I introduce convolutional layers, uncertainty predictions, and my *input quality based dropout* layer (qDrop) to the architecture, which proves to outperform the architecture with only bLSTM layers. The final architecture has no fully connected layer and allows to determine the impact of individual time series steps on the prediction. This last update also removes a hyper-parameter by learning the noise function for the heteroscedastic uncertainy in separate learning run with a new loss function.

Stefan Oehmcke

Deep Learning of Virtual Marine Sensors

Zusammenfassung

Das Verständnis unserer Meeresökosysteme ist wichtig, um die Umwelt zu schützen und die nachhaltige Nutzung von Ressourcen zu fördern. Insbesondere sind Küstenprozesse interessant, da die Küstenregionen den größten Seeverkehr und das meiste Meeresleben aufweisen. Diese Prozesse zu beobachten ist eine schwierige Aufgabe aufgrund der harten Unweltbedingungen und der Größe des abzudeckenden Bereichs. Um diese Aufgabe zu lösen, werden robuste Sensoren benötigt, welche Stürme, Salzwasser und den täglichen Verschleiß der Gezeiten bewältigen können. Durch die wachsende Sensorinfrastruktur sind die anfallenden Datenmengen nicht mehr manuell verarbeitbar und die Wahrscheinlichkeit, dass ein Sensor ausfällt, steigt ebenfalls.

Diese Arbeit stellt einen Ansatz vor, wie virtuelle Sensoren basierend auf maschinellem Lernen erstellt werden können, um nicht funktionierende physikalische Sensoren zu ersetzen. Diese virtuellen Sensoren werden mit Sensordaten von der *Time Series Station Spiekeroog* (TSS) und dem *Biodiversity-Ecosystem Functioning across marine and terrestrial ecosystems* (BEFmate)-Projekt im Wattenmeer (Deutsche Bucht) trainiert. Im ersten Teil meiner Arbeit erkläre ich zunächst die Daten und deren Vorverarbeitung. Als nächstes wird eine unüberwachte Extremereigniserkennungsaufgabe auf den TSS-Daten mit einer nachfolgenden Expertenauswertung präsentiert. Dann schlage ich eine Imputationsmethode für längere, fortlaufend fehlende Werte vor, wie sie in unseren Datensätzen vorkommen. Das Verfahren verwendet als ersten Schritt eine lineare Interpolation, aber bestraft diese interpolierten Werte basierend auf der Länge der Lücken mit einem k-nächste-Nachbarn-Verfahren (penalized DTWkNN ensemble). Im zweiten Teil entwickle ich eine neuronale Netzwerkarchitektur, um ausgefallene Sensoren aus dem BEFmate-Projekt zu modellieren. Die Grundlage ist ein bidirektionales rekurrentes neuronales Netzwerk mit Langzeit-Kurzzeitgedächtnis (bLSTM), welches meine zeitliche Dimensionalitätsreduktionsmethode exponential piecewise approximate aggregation (exPAA) verwendet. Dann führe ich Schichten mit Konvolutionen, Unsicherheitsvorhersagen und meine auf Eingangsqualität basierende Dropout-Schicht (qDrop) in die Architektur ein, welche bessere Ergebnisse erziehlt als die Architektur nur mit bLSTM-Schichten. Die finale Architektur hat keine vollständig verbundenen Netzwerkschichten und erlaubt es, die Auswirkung einzelner Zeitreihenschritte auf die Vorhersage zu bestimmen. Außerdem wird ein Hyperparameter entfernt, indem die Rauschfunktion für die heteroskedastische Unsicherheit in einem separaten Lernlauf mit einer neuen Verlustfunktion gelernt wird.

Contents

Acknowledgments

First, I would like to thank my supervisors. Oliver Kramer, who sparked my interest in computational intelligence and motivated me with his helpful feedback to continue working on my research. Oliver Zielinski, who showed me invaluable, non-computer science perspectives and insights into the marine domain and academia. I highly appreciate that they gave me the freedom to explore my ideas, but also provided the right impulses when I needed them. Both of them have always had an open ear for me and found time in their busy schedules.

Further, I would like to thank my committee consisting of Korbinian Riedhammer, Jürgen Sauer, Jörg Bremer, and my supervisors. I am also grateful for all the ideas, feedback, and nice work environment from the *computational intelligence* group and the *marine sensor systems* group. Finally, I want to thank my friends and family for their emotional support and the beautiful distracting moments away from my research.

This work was funded by the *safe automation of maritime systems* scholarship. The BEFmate project (Biodiversity – Ecosystem Functioning across marine and terrestrial ecosystems) was funded by the Ministry for Science and Culture of Lower Saxony, Germany under project number ZN2930.

1 Introduction

1.1 Motivation and Problem Statement

The observation of coastal processes is important to understand marine ecosystems [6, 33]. This in turn helps us to protect the environment and make resource management sustainable [93]. For example, monitoring the salinity of an area indicates what type of life can exist there, and a shift of that property signalizes that the habitat is also changing. Another example is the early detection of algae blooms in coastal areas, as they are dangerous to humans and fish populations. To cover such areas, an array of sensors is required. They must meet requirements, such as reliability, robustness, and minimal need for maintenance to withstand the challenging environmental influences. For instance, these influences can be storms, saltwater, and the daily wear caused by the tides.

As the infrastructure of sensor arrays grows and the resolution of sensors increases, more and more data are produced. It is not practical to process these data manually. The properties of the data are also challenging, be-

cause they usually contain a high degree of noise, outliers, and missing values. Further, there are non-trivial inter-sensor relationships, since the context of the seas always changes due to tidal or seasonal influences. Although approaches are available to deal with some of these challenges, they often do not work with large datasets in a feasible time frame or lack stability in their output performance.

Here, I mainly work with two real-world marine datasets, which provide sensor data for a part of the German Wadden Sea. The first dataset originates from the Time Series Station Spiekeroog (TSS) [6, 120]. It is a long term sensor platform that is exposed to strong tidal influences. The second dataset consists of sensor measurements from the Biodiversity-Ecosystem Functioning across marine and terrestrial ecosystems (BEFmate) project [8, 9]. The BEFmate sensors are located on and around twelve artificial islands in front of Spiekeroog. Their close proximity to each other allows for a combination of these datasets. Both datasets present their own challenges, but are invaluable for the marine research of the Wadden Sea.

In my data, a flow sensor of the BEFmate project failed and needs to be replaced. When such a sensor fails completely, it may not be repairable and thus diminishes the quality of the sensor array to which it belongs. The replacement should be a model based on data of surrounding sensors. Such a modeling approach is more flexible and transferable than a physical or numerical modeling approach, which would require expert knowledge of the domain and expensive laboratory experiments [36]. Numerical models are also not feasible, because they are highly specialized to one physical property and transferring to another property is impractical and costly. To the best of my knowledge, there is no method capable of creating such a model in this or any similar application, which includes preprocessing and imputation of missing data, while remaining reliable and scalable on large datasets.

1.2 Approach - Machine Learning

I use machine learning, a sub-field of artificial intelligence, to approach the challenges posed by the datasets. A machine learning algorithm builds a hypothesis or model f that is able to make a decision about a previously unseen pattern with particular features based on information from observed patterns and decisions [15, 34, 57, 100]. It learns to recognize patterns and stores these information in the model. This model is part of the hypothesis space or model set F with all possible models in an application, so $f \in F$. A model is given a matrix X of n input vectors or patterns $X := (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$. One input vector consists of d features $\boldsymbol{x} := (x_1, \ldots, x_d)^T$ in feature space $\boldsymbol{x} \in \boldsymbol{X}^d$. In our case, the input features are mostly sensor measurements that are continuous in nature. This results in a real numbered feature space ($\boldsymbol{X}^d = \mathbb{R}^d$). All n input vectors produce a decision $f(X) := \hat{Y}$, which is called target output and is a matrix $\hat{Y} := (\hat{\boldsymbol{y}}_1, \ldots, \hat{\boldsymbol{y}}_n)$ with d' dimensional vectors $\hat{\boldsymbol{y}} := (\hat{y}_1, \ldots, \hat{y}_{d'})^T$ in target space $\hat{\boldsymbol{y}} \in \boldsymbol{Y}^{d'}$. These outputs approximate the true targets $f(X) \approx Y$. For every input and output vector there also exists a true target output $\boldsymbol{y}$, which is not always known. Models are created using learning algorithms, such as artificial neural networks, that are inspired by the neural networks in the brain. They learn an inner representation of a training set of patterns to produce output vectors for an unseen test set.

The nature of a decision depends on the existing learning problem. There is a manifold of learning problems, such as classification, regression, ranking, or dimensionality reduction. Classification is a long time topic in machine learning. Here, the target outputs are k qualitative classes $y \in \{c_1, \ldots, c_k\}$ and a model is called classifier. For example, the classification of fish into classes of species $y \in \{\text{pike}, \text{perch}, \text{goldfish}, \ldots\}$ by examining features such as weight, length, color, and number of fins. In regression, the targets are numeric, i.e. $\boldsymbol{y} \in \mathbb{R}^{d'}$ and models are referred to as regressors. Examples for regression applications are the forecast of power output

for wind parks with a horizon of a few hours [145] or the prediction of food consumption of fish populations [109]. Ranking problems suggest an order for a set of patterns. This is useful for data mining tasks such as the prioritization of search engine results. The feature space can be very large, which complicates the exploration or visualization of information, but dimensionality reduction handles this problem. Here, the model output is a transformation of the feature space with d input dimensions to a lower representation with d' target dimensions, whereby $d' < d$.

Learning scenarios define what kind of input sets are available. Supervised learning assumes that there are patterns with their respective true targets known to the model. These pairs of patterns and true target outputs are given to the learning algorithm during training to create the model. Typical learning problems for supervised learning are classification, regression, or ranking. Unsupervised learning has no information about the true target output and in effect only trains with the patterns. This scenario is often used to find structures in data for clustering problems or to reduce the dimensionality of the feature space. Semi-supervised learning combines the two previous scenarios. There are a few pairs of patterns and true target outputs to train the model, but more unlabeled pattern are available to understand the underlying structure. This scenario occurs when true targets are hard to acquire, but patterns are easy to obtain.

Together, learning scenarios and problems define the learning task. The results on that task are highly dependent on the quality and quantity of data available as well as the employed algorithm for processing the data. These are the basis for a model and guarantee a correct result to a certain degree [16]. Concrete applications for machine learning always arise, when the human being either does not want to or cannot process the data. Hastie *et al.* [57] emphasize this by stating that today the amount of data is enormous, but can no longer be analyzed or interpreted manually.

Performance metrics are indicators of a models prediction performance.

The used metrics vary between the learning tasks. For example, a regression task could use the absolute error between the true and approximated targets, which is then used to find the model that minimizes the average error on a set with n test patterns:

$$\underset{f \in F}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i| \,. \tag{1.1}$$

Similarly, in a classification task, we seek a classifier that maximizes class prediction accuracy. Unsupervised methods cannot rely on true targets and thus measurements such as the preservation of neighborhoods for dimensionality reduction or other data characteristics, such as correlations are required.

1.3 Contributions

In this work, I offer machine learning and data analysis solutions for a coastal observation application to answer the question:

> How can the broken flow sensor of the BEFmate project be replaced by a machine learning model based on the surrounding sensor data of BEFmate and TSS?

By answering this, I hope to allow a better observation of coastal processes as well as giving a procedure to replicate such models for similar applications. This includes the following intermediate steps:

1. General preprocessing
   Creation of a coherent dataset and correcting drifts in data;
2. Filling missing values
   Introducing and applying a novel time series imputation method;
3. Designing a deep learning architecture
   Iterating and improving neural network architectures.

My data-driven solutions are less computationally expensive and less expert knowledge is required than in classical physical or numerical modeling approaches. I develop these solutions in a problem-oriented way, but also ensure that they are generalizable to similar problems and applications.

1.4 Structure of This Thesis

In Part I, the focus lies on the dataset and several preprocessing steps. First, I introduce the marine datasets and describe the general preprocessing steps applied to it in Chapter 2. Then, I show in Chapter 3 that automated extreme event detection is possible with an unsupervised machine learning approach, which is evaluated by experts. In Chapter 4, I present a novel imputation algorithm that is able to replace multivariate missing data.

Part II includes work on the virtual sensor with deep learning. Initially, deep learning is explained in Chapter 5. I present the basic design of my architecture in Chapter 6 based on bidirectional long short-term memory network (bLSTM). Thereafter, I improve this architecture with the addition of convolutional layers, uncertainty predictions, and by regarding the known input quality in Chapter 7. Finally, Part III gives a conclusion of this work and ends with an outlook to possible future research questions in Chapter 8.

Part I

Dataset and Preprocessing

After an introduction to the research problem and the approach to solving it through data-driven machine learning, Part I focuses on the data of the application. I provide general and detailed information about the datasets. Further, I explain the employed preprocessing steps as well as an exemplar task and propose an imputation method especially designed for the properties of these datasets. This is an important basis for creating a virtual sensor in the next part.

There is no such thing as clean or dirty data, just data you don't understand.

Claudia Perlich

2 Marine Datasets and Preprocessing

2.1 Introduction

Interesting problems arise from real data. Artificial datasets can help to isolate certain aspects of a problem, but this requires detailed knowledge about the domain of the real data. Customers want modeling solutions based on real data. For example, an energy company rather wants predictions about the power output of their real offshore wind turbines than on artificial test data; or marine researchers want sensor data predictions based on their in-situ sensor rather than a physical miniature model.

This work focuses on two datasets from the marine domain which have been measured for scientific purposes, such as the monitoring and understanding of natural dynamics in the Wadden Sea. The first dataset consists of measurements of the TSS and is presented in Section 2.2. The second dataset is described in Section 2.3, where the sensor measurements are taken from a nearby biodiversity experiment by the BEFmate project. I later combine both datasets to one coherent set, which is detailed in Sec-

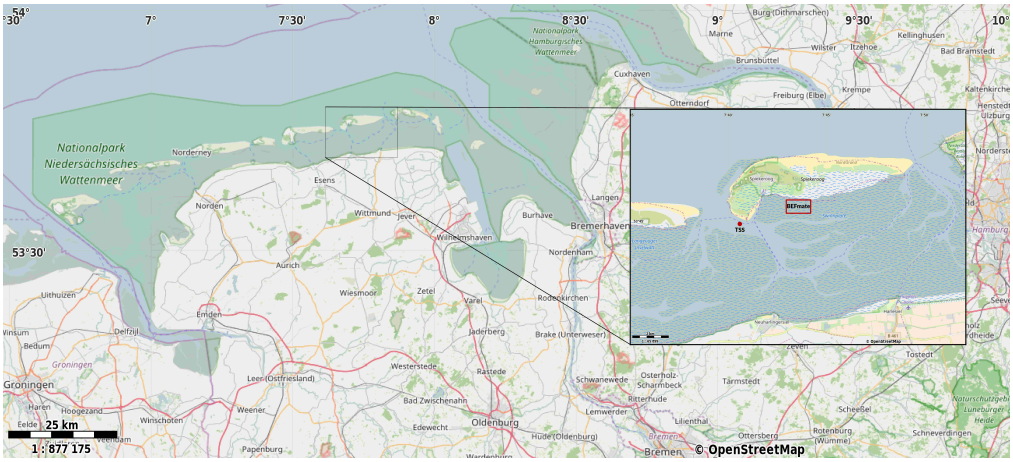


Figure 2.1: Position of TSS and artificial islands of BEFmate in the Wadden Sea. Map data copyrighted OpenStreetMap contributors and available from `https://www.openstreetmap.org`.

tion 2.4. Figure 2.1 shows the position of both data sources. The target sensor for the final virtual sensor model is explained at the end of that section. Thereafter, I describe the required preprocessing steps in Section 2.5 and conclude the chapter with Section 2.6.

This chapter is partly based on the following published papers:

> Oehmcke, S., Zielinski, O., and Kramer, O. (2015). Event detection in marine time series data. In *Advances in Artificial Intelligence - Annual German Conference on AI (KI)*, pages 279–286. Springer,

> Oehmcke, S., Zielinski, O., and Kramer, O. (2017b). Recurrent neural networks and exponential PAA for virtual marine sensors. In *International Joint Conference on Neural Networks (IJCNN)*, pages 4459–4466. IEEE.

2.2 Time Series Station Spiekeroog

The TSS is a stationary platform with a multitude of sensors installed [6, 105, 120]. The platform is positioned between the islands of Langeoog and Spiekeroog (53°45′1.0″ N, 7°40′16.3″ E) in the North Sea (see Figure 2.1). It has been in service since 2002 and is located in a tidal channel, resulting in stronger tidal currents than average in this area. Because of this, the TSS requires a robust structure and reliable hardware to measure continuously throughout the year. Its structure is a 35.5 m long pole that is set 10 m into the ground. On top of the pole are two laboratory containers at 7 m above mean sea level to conduct on-site experiments. Air temperature, pressure, and humidity as well as wind speed and direction are measured from the roof of these containers. The power supply is self-sustaining with energy from solar and wind power as well as reserve batteries and a gas powered generator. Figure 2.2 shows the pole, the container, the wind turbine, and a subset of the above sea level sensors. Underwater, there are measuring points at 4, 5.5, 7.5, 9.5, and 11.5 m below mean sea level. At these points, water temperature, conductivity, pressure, salinity, and several other chemical compounds are monitored. The sampling rate is standardized to one measurement per minute. A computer system manages the TSS, which caches the sensor measurements and periodically sends them to a server. More details about the sensors can be found in Table A.1.

2.3 BEFmate Artificial Island Project

My second dataset is based on data from the BEFmate project that studies the biodiversity ecosystem functions in marine and terrestrial environments [8, 9]. In one experiment of the BEFmate project, they built twelve artificial islands in the back-barrier tidal flat of the island Spiekeroog

Figure 2.2: The TSS on the left side and the now malfunctioning flow sensor on the right side.

(53°45′31″ N, 7°43′30″ E). These islands extend from northwest to southeast for over 810 m, whereby each 2 m × 6 m island consists of twelve steel cages in four different elevations. The construction was finished in 2014. In contrast to the TSS, the mean high water level is only ∼ 0.8 m. Each steel cage contains a plot either filled with sediment, transplanted parts of salt marshes, or left untouched. To protect the plots from the average waves, an upward-widening conical steel plate is installed around the plots. One of the main objectives of the BEFmate experiment is the observation of vegetation growth in these islands. To support and explain these observations, the islands are equipped with an array of abiotic sensors to observe environmental properties. I use temperature, wind speed and direction, air pressure, moisture, energy, brightness, and rainfall measurements from different positions around the islands. For example, six of the temperature sensors are installed directly on the surface of the plots. The hardware

Figure 2.3: Picture of the 12th BEFmate island at high tide (left) and the 11th island at low tide (right).

includes: six HOBO® U20L water level loggers, six DEFI-T temperature loggers, an RBRduo TD | wave sensor, and a meteorological sensor station. The sampling rates for these sensors range from one per minute to one per ten minutes.

2.3.1 Target Sensor: Faulty Flow Sensor

The sensor system of BEFmate also included a flow sensor (SEAGUARD® RCM) as seen in Figure 2.2, which does not work anymore. Our objective for Part II is to use this sensor as target for a virtual sensor model. It observed the *Speed*, *Temp* (temperature), *Conductivity*, and flow *Direction* of the water. In the following I will always refer to these sensors, if the spelling is capitalized as in the previous sentence. This flow sensor was positioned between the artificial islands at 53°45′29″ N, 7°43′17″ E. Measurements were not performed at low tide as the sensor was dry at these times. Ever since it failed at the end of 2015, it could not be replaced physically. A virtual sensor replacement would be able to approximate data that would otherwise be lost to researchers. This virtual sensor would be based on the neighboring sensors and help to keep an overview of coastal processes in that area.

Figure 2.4 displays the five measured properties of the flow sensor. In ad-

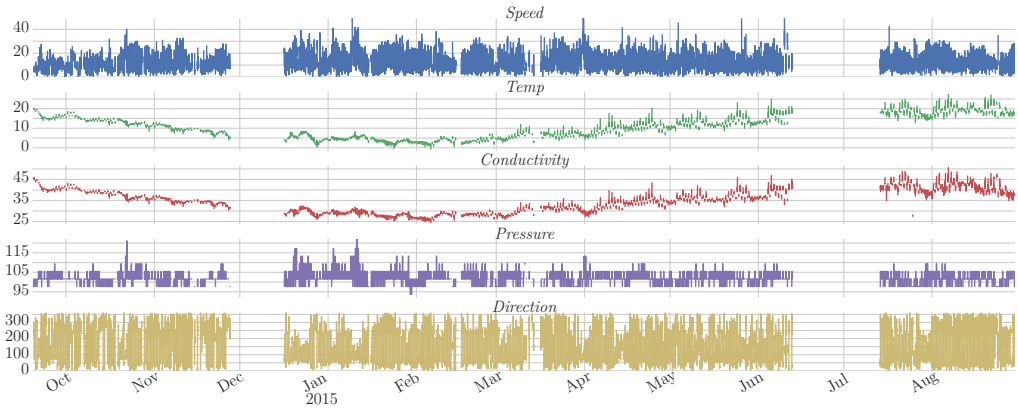


Figure 2.4: Overview of all five time series of the flow sensor.

dition to missing values at low tide, there are two bigger gaps in December 2014 and July 2015. These time steps are lost for building or evaluating a model. Seasonal effects are visible for *Temp* and *Conductivity*. Otherwise, there is no trend visible in the observed time frame. Most dynamics in the time series result from the tides.

Count and density plots of the flow sensor are presented in Figure 2.5. I made a distinction between data for training and testing to highlight differences. These are especially apparent for *Temp* and *Conductivity*, because of their dependence on the seasons. The three other sensor properties have relatively constant densities over the training and test set. At first, the *Pressure* measurements appear to have five density centers, but this is due to the low measuring resolution of the sensor. In reality, the density is more similar to the *Speed* measurements, which have a single center and a skew to the right. For the *Direction* measurements two density centers exist since the water either originates from the North Sea or flows back into it.

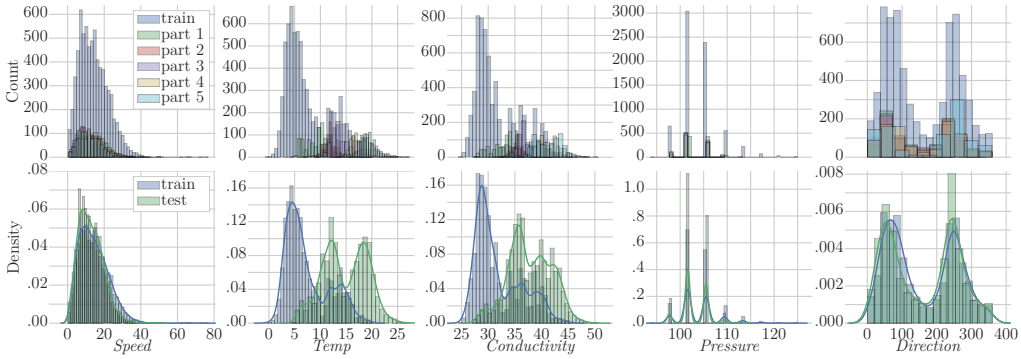


Figure 2.5: Distribution plots of the faulty sensors. The upper plots compare the counts in the different test parts to the training data, while the lower plots show the density of training and test data.

2.4 Combined Dataset

The combined TSS and BEFmate dataset contains measurements from 2014-09-18 15:00 to 2015-08-30 22:00. This amounts to 49 867 time steps from 57 different time series. Whereby 18 sensors are provided by the TSS and 38 by the BEFmate project. A combination is justified as there are strong correlations due to their close proximity, as shown in Figure 2.6. I also include an approximation for the height of the tide based on the time and space model by Paul Schureman [128]. The first data points for the flow sensor are at 2014-09-19 05:40. A total of 11 633 time steps are available for the flow sensor because it was usually not measured at low tide.

2.5 Preprocessing

To create a coherent combined TSS and BEFmate dataset, several preprocessing steps are necessary. These steps are not applied to the target flow sensor, because they would change scale and behavior of values. Data from the TSS and the BEFmate projects weather station have minute-wise mea-

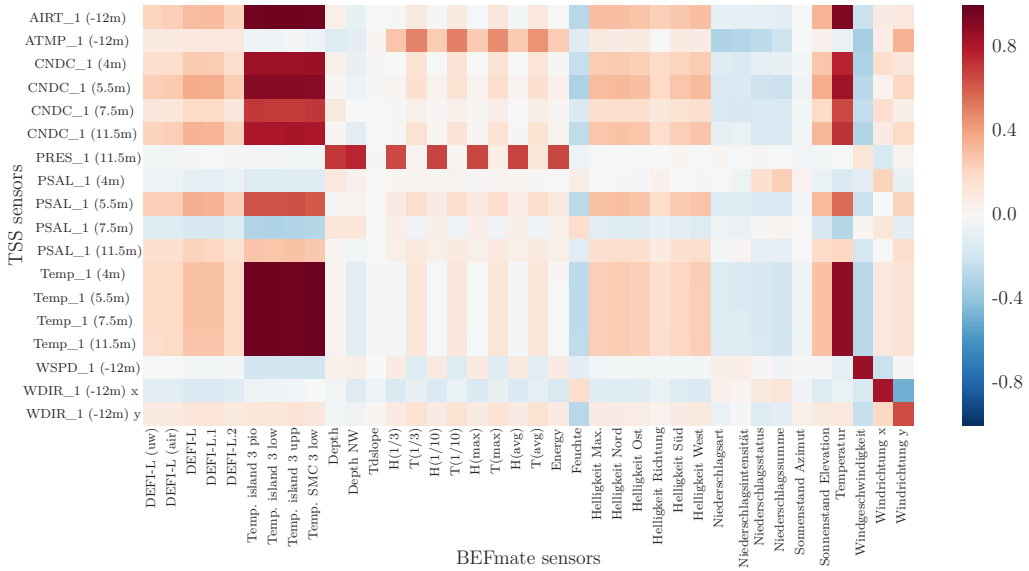


Figure 2.6: Correlation matrix of TSS and BEFmate dataset. More intense colors indicate stronger correlations.

surements, while the rest samples every ten minutes. I unified the data by down-sampling the minute-wise measurements. To that end, first the concerned series are smoothed with a moving median of ten minutes. Then, only every tenth value is kept, which results in a number of time steps equal to the other series.

To stabilize the variance of measurements that seem to have a non-Gaussian distribution, box-cox transformations [17] are applied [67]. Measurements for light, energy, rain, and depth are transformed. These power transformations output y_i^{boxcox} for an input feature x_i are defined with a parameter λ:

$$y_i^{boxcox} := \begin{cases} \frac{x_i^{\lambda}-1}{\lambda} & \text{for } \lambda > 0 \\ \log(x_i) & \text{for } \lambda = 0 \end{cases} . \tag{2.1}$$

To avoid division by zero errors, all affected series get their absolute minimal value added as well as a small term of $1e{-}6$. The parameter λ is chosen

automatically by maximizing the adapted log-likelihood function [17]:

$$llf(\lambda) := (\lambda - 1) \sum_{t=1}^{n} \log(X_{t,i}) - \frac{n}{2} \log\left(\frac{\sum_{t=1}^{n} (Y_{t,i}^{boxcox} - \overline{y}^{boxcox})^2}{n} \right), \quad (2.2)$$

with $\overline{y}^{boxcox}$ being the average boxcox value for the given λ.

2.5.1 Removing Drifts and Shifts

Biofouling, the accumulation of such biomass, can be a problem, as the marine environment is inhabited by microorganisms and algae [44, 139]. In fact, the eight raw conductivity and salinity time series from the TSS are afflicted by this. Salinity is affected because it is calculated using the conductivity. The result is a negative drift of these time series, which do not reflect the actual physical properties. Moreover, shifts occur when a cleaning of the affected sensors is conducted. Together, the drifts caused by biofouling and the shifts caused by the necessary cleaning of the sensors have a negative effect on the quality of the data. If the drifts would remain in the time series, removing this feature would be the only valid option.

First, I remove the point outliers to create a more stable time series for the drift correction. A point outlier is a single value of a time series that differs significantly from previous and following time step. Only point outliers of the affected time series are removed because most machine learning algorithms are robust against outliers. To that end, I calculate a moving standard score for each time step with the distribution coming from a centered time frame of size 60 and 30 minutes, respectively. Then, an isolation forest [91] (see Section 3.3.2) is trained to acquire an outlier score. If the deviation to the median of scores exceeds three times the standard deviation of all outlier scores, the value is removed.

Now, the shifts are detected to gain reference points for the drift correction. There is no information about when the sensors are getting cleaned,

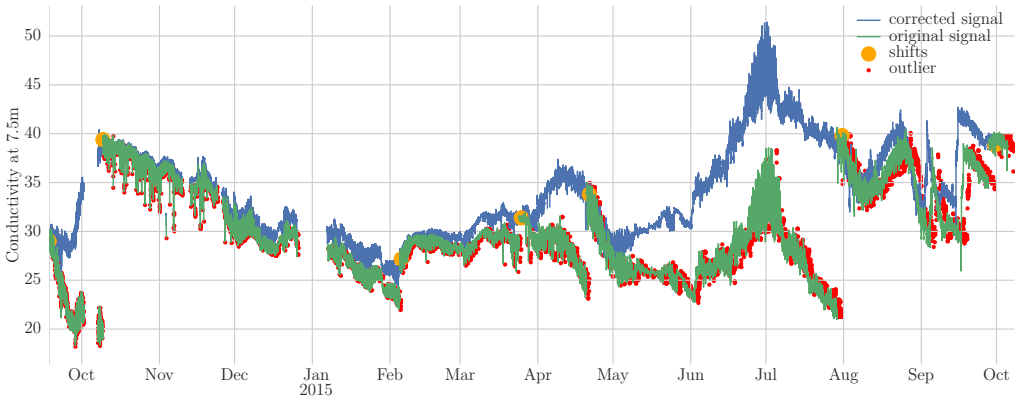


Figure 2.7: Exemplary original and drift-corrected series for the conductivity sensor at 7.5m height. Also highlighted are detected outliers and shifts.

but I know when general maintenance has been performed, which may include cleaning. This reduces the number of time steps that must be checked for shifts. As a reference, I calculate the moving median and moving median absolute deviation of one tidal cycle to create a smoothed time series. A shift is then detected if at any time during maintenance the difference to the previous time step is bigger than three times the moving absolute deviation. To ensure that the detected shift is not a natural occurrence, it is compared to the more stable temperature time series at the same measuring height. If I detect a shift simultaneously at the affected and the reference time series, I ignore it as a false positive.

Lastly, the drifts are corrected by adding a decaying linear offset between two shifts. This offset is the difference between the following shift and the median of the last values that were not shifted. The last values before a shift get the full offset added, while shortly after a shift, this added offset is smallest. This follows the intuition that the most accurate values are present after a cleaning and the sensor is then slowly overgrowing. Figure 2.7 shows the original and the corrected signal.

2.6 Conclusion

In this chapter, I first introduced the two datasets from the TSS and the BEFmate project, which are later combined to one coherent dataset. One of the sensors, a flow sensor of the BEFmate dataset, malfunctioned permanently will later be modeled as virtual sensor. This broken sensor shows interesting characteristics, for example, it only measured at high tide. Overall, the data are challenging due to natural influences such as the strong tidal currents or seasonal fluctuations. Preprocessing was necessary to create the same time resolution as well as the correction of drifts caused by bio-fouling. Finally, the resulting data are now usable for machine learning algorithms.

3
Extreme Event Detection

3.1 Introduction

I introduced the marine datasets in the last chapter. In general, the growing infrastructures of information and sensor systems create data that are no longer manageable by humans. The diversity of the sensors from the TSS as well as the redundancy over different heights is crucial for the creation of valuable environmental data. Even a small subset of 16 sensors is hard to comprehend entirely over a longer period. When time series datasets are analyzed manually, short but interesting events might be overlooked.

In this chapter, I present an automatic detection approach for extreme events in the TSS data. These events could be storm surges or unusual sensor value movements [158]. Extreme events are loosely defined as points or periods in time that are vastly different from regular data. This informality in the definition often creates disagreements in experts, because they value different properties in irregular data. By providing automatic

detection, experts can focus on analyzing these events instead of searching for them. I employ the local outlier factor (LOF) algorithm and extend the top-k ranking with a time component as my data are time series. In effect, my proposed approach finds events that show positive results in an expert evaluation. The first results could be improved through dimensionality reduction with isometric mapping (ISOMAP) [138]. I also introduce a new visualization based on the ISOMAP representation.

In the past, many event detection methods have been introduced, but they are seldomly applied to the marine domain. An overview of temporal outlier detection methods are the surveys by Gupta *et al.* [55] and Chandola *et al.* [24]. A comparison of novelty detection methods on synthetic time series data is given by Modenesi *et al.* [99]. Auslander *et al.* [5] employs outlier detection in the maritime domain for video analysis. They highlight the different conditions under which local or global algorithms perform better. Other work focuses on the anomalies in vessel movement, where suspicious movements should be recognized as anomalies. A system grounded on a rule- and motif-based framework that can learn at different granularity levels is proposed by Li *et al.* [87]. A similar framework by Riveiro *et al.* [121] employs a self-organizing map method for event detection together with an intuitive user interface. Nevertheless, sensing extreme events for sensor platforms such as the TSS has not been researched to the best of my knowledge.

I will begin by giving a general definition of time series events is given in Section 3.2. In Section 3.3, I present specific methods for event detection. The experiments with marine data from the Time Series Station will be shown in Section 3.4. Finally, conclusions are presented in Section 3.5.

This chapter is based on the following published paper:

> Oehmcke, S., Zielinski, O., and Kramer, O. (2015). Event detection in marine time series data. In *Advances in Artificial Intelligence - Annual German Conference on AI (KI)*, pages 279–286. Springer.

3.2 Extreme Event Definition for Time Series

There exists no universal definition for extreme events [24]. Depending on the application other words are used. Terms such as outlier, anomaly, or novelty are common. In this marine application, I use the term extreme event or just event, because the other terms are closer associated with errors in the measurement processes than an incident in the observed process.

The binary classification problem statement is well suited for the event detection task. The true target output y_t is predicted with a model $f(\boldsymbol{x}_t)$. There is one class for normal and one class for event data represented in our target $y_t \in \boldsymbol{Y} = \{normal, event\}$. The input to this model is a pattern $\boldsymbol{x}_t := (x_1, \ldots, x_d)^T \in \boldsymbol{X} = \mathbb{R}^d$ from some time step $t \in (1, \ldots, n)$ of the series. An event is then detected if the outlier function o_{score} is exceeding a certain threshold γ at a time steps t':

$$\max_{t' \in (1,\ldots,n)} o_{score}(\boldsymbol{x}_t, \boldsymbol{x}_{t'}) > \gamma \ . \tag{3.1}$$

Instead of a hard classification, there can be a soft classification with the predicted target output $\check{y}$:

$$\check{y} := \max_{t' \in (1,\ldots,n)} o_{score}(\boldsymbol{x}_t, \boldsymbol{x}_{t'}) \ , \tag{3.2}$$

with a higher value meaning a higher affiliation with the outlier class. What kind of events are found is highly dependent on the algorithm behind the outlier function, but also on the chosen frame of the dataset and the threshold value. With domain knowledge, finding these parameters is easier.

Events can be found in a *univariate* or *multivariate* way. Univariate events can be detected in only one of the d features. Since the other features hold no information about the event, the effective dimensionality is one. Moreover, multivariate events are harder to find, because not only

one feature is anomalous, but multiple at once. For the TSS dataset, I consider both variants.

By including δ observations into one pattern, I achieve time dependency. To keep compatibility to most outlier score functions the resulting matrix $X_{t-\delta+1:t}$ is flattened to a vector $\boldsymbol{x}_t^\delta$:

$$\boldsymbol{x}_t^\delta := \mathsf{flatten}(X_{t-\delta+1:t}) := (\boldsymbol{x}_{t-\delta+1}^T, \ldots, \boldsymbol{x}_t^T)^T \ . \tag{3.3}$$

The period length δ needs to be selected wisely, because if chosen too small, an event might not be detected, if chosen to big, finding the exact point of occurrence is problematic.

3.3 Event Detection Methods

In the following, I describe the employed event detection methods in this work.

3.3.1 local outlier factor and k-top-time Method

As the extreme events are not labeled in my dataset, I apply an unsupervised approach for the outlier score. Introduced by Breuning *et al.* [20] in 2000, the LOF established itself as a standard event detection method. The main idea is to compare an unseen pattern $\boldsymbol{x}$ to the training set and calculate how densely populated the feature space around that pattern is and compare it to its k neighbors density. If the k neighbors of $\boldsymbol{x}$ have a higher density, the more likely the pattern is an event. With the k nearest neighbors function $\mathrm{NN}_k(\cdot)$, the outlier score for LOF is:

$$\mathrm{LOF}(\boldsymbol{x}) := \frac{1}{|\mathrm{NN}_k(\boldsymbol{x})|} \cdot \sum_{\boldsymbol{x}' \in \mathrm{NN}_k(\boldsymbol{x})} \frac{\sum\limits_{\hat{\boldsymbol{x}} \in \mathrm{NN}_k(\boldsymbol{x})} \mathrm{rd}_k(\boldsymbol{x}, \hat{\boldsymbol{x}})}{\sum\limits_{\bar{\boldsymbol{x}} \in \mathrm{NN}_k(\boldsymbol{x}')} \mathrm{rd}_k(\boldsymbol{x}', \bar{\boldsymbol{x}})} \ . \tag{3.4}$$

The reachability-distance $\mathrm{rd}_k(\boldsymbol{x}, \boldsymbol{x}')$ is the greater value of either the distance between $\boldsymbol{x}$ and $\boldsymbol{x}'$ or the k-distance $\mathrm{dst}_k(\boldsymbol{x}')$:

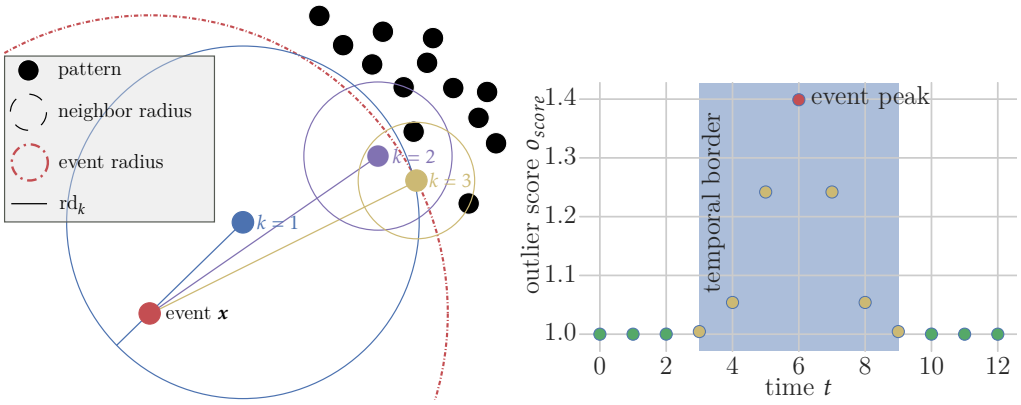
$$\mathrm{rd}_k(\boldsymbol{x}, \boldsymbol{x}') := \max(\mathrm{dst}_k(\boldsymbol{x}'), \Delta(\boldsymbol{x}, \boldsymbol{x}')) \ . \tag{3.5}$$

In most cases, the Euclidean distance Δ_{eucl} in $\mathbb{R}^d$ is used as distance measure Δ for two patterns $\boldsymbol{x}$ and $\boldsymbol{x}'$:

$$\Delta_{eucl} := \sqrt{\sum_{i=1}^{d} (x_i - x_i')^2} \ . \tag{3.6}$$

The k-distance $\mathrm{dst}_k(\boldsymbol{x}')$ is defined as distance of $\boldsymbol{x}'$ to its k-th neighbor. This way, the distance is more stable because, if $\boldsymbol{x}$ is part of the neighborhood of $\boldsymbol{x}'$, the distance is still the furthest neighbor of $\boldsymbol{x}'$. Figure 3.1a shows the reachability-distance by comparing an event $\boldsymbol{x}$ to its neighbors.

The resulting outlier score of LOF scales between zero and a positive open end. If the score is lower or equal to one, it belongs to the inliners. But if the score is greater than one, it is not clear when an extreme event is happening. The appropriate threshold γ is hard to define. Another option is to apply a top-k method that registers the k highest scores as outliers [72]. The advantage of this method is that I only have the most likely events and can then ask experts to check them. However, interesting events often happen over multiple time steps, which can be a problem if most of the k top events are actually the same event. I create an adapted method to deal with this problem, called top-k-time. My adapted method builds a temporal border around the highest score to block patterns within from being chosen. In Figure 3.1b this border is visualized. I repeat this with the next highest value until all k events are found.



(a) The reachability-distance visually explained for $k = 3$ in a 2-dimensional feature space. The first neighbor uses the k-distance and the other the normal distance measure.

(b) Example of my top-k-time approach. A high outlier score blocks other pattern from being detected. Here, the border is three time steps on each side of the detected pattern.

Figure 3.1: Reachability distance and k-top-time method explained.

3.3.2 Isolation Forest

Isolation forest by Liu *et al.* [91] is an ensemble method, which combines several isolation trees for its outlier prediction. The main idea assumes that outliers are few and different in regard to the rest of data and thus can be isolated. To build a binary isolation tree, we create nodes by successively drawing a random feature and split value, which divide the dataset into two new nodes. A tree grows as long as there are more than one pattern in a node and a prior set maximal depth is not reached. The outlier score for isolation forest is derived by the amount of traversed edges $h(\boldsymbol{x})$ from the root, which are needed to isolate a pattern vector $\boldsymbol{x}$. The intuition behind this is that outliers require fewer edges than normal patterns, making them easier to isolate. To normalize $h(\boldsymbol{x})$, we relate it to the average traversed edges for n patterns:

$$c(n) := 2H(n-1) - \frac{2(n-1)}{n}\,, \tag{3.7}$$

with harmonic number $H(i)$. Accordingly, the outlier score is:

$$o_{score}(\boldsymbol{x}, n) := 2^{-\frac{E(h(\boldsymbol{x}))}{c(n)}} , \tag{3.8}$$

where $E(h(\boldsymbol{x}))$ the average traversed edges from all isolation trees.

3.4 Experimental Analysis

In this section, a machine learning event detection approach is applied to a new marine application with part of the TSS dataset and evaluated by experts.

3.4.1 The Dataset

For these experiments, I employ the TSS dataset in the time from 2014-10-10 to 2015-04-13. There is a high probability for extreme events, because it is recorded in the storm season. To reduce the number of patterns from 267 840 to 17 856, I apply a centered rolling median of 15 minutes $\delta^{roll} = 15$ and retain only every 15th pattern, which also reduces noise in the data. This is inspired by Basu and Meckesheimer [10] and a 15 minute sampling rate is not a high loss in information and is still able to capture events of interest. I use a set of features that domain experts selected w.r.t. my event detection task. This set consists of 16 sensor values that observe water properties such as temperature, salinity, and pressure, but also air properties such as temperature, wind direction, and wind speed. Together with nine time features, I have 25 features. This dataset is affected by missing data, with 29.6% (5285) of the data completely or partially missing because of maintenance or connection failures. When less or equal than ten values are missing in a row, these missing values are interpolated with the piecewise cubic Hermite interpolation polynomial method [41].

There are nine time features, because if time is mapped directly into one linear feature, repeating structures such as the day-night-cycle can not be represented. Therefore, I mapped minutes, hours, days, and months to circles to portray the circulation of time (e.g. 23:00 and 01:00 have a difference of two hours instead of 22). Example for hour h: $h_{sin} := \sin\left(\frac{h}{24} \cdot 360°\right)$ and $h_{cos} := \cos\left(\frac{h}{24} \cdot 360°\right)$. In the same way, the wind direction, which is represented as degree is mapped to a circle to avoid problems when the values switch between 360 and one degree.

3.4.2 Expert Survey for Label Acquisition

I need labeled data to evaluate event detection methods, otherwise my algorithms only produce an arbitrary clustering of data with no verification. As the complete dataset would be to big for a human to label, I show a group of five marine researchers a small selection of 60 patterns. The patterns with the individual sensors (unchanged features) are displayed in a web-based survey that I programmed with accessibility in mind. An expert can specify if a pattern is an event and also add a small text, if necessary.

The 60 pattern consist of the top-20-time results of three LOF approaches, each with a different method to handle features: The first method produces the LOF score with the untouched features ($d = 25$), which I refer to as the raw value method. The second method highlights strong differences, here referred to as rolling mean method. It alters the sensor based features by individually taking the squared difference to the centered rolling mean of a period δ. This increases the number of features to $d = 41$, because the untouched are also part of the feature set. Finally, the third method is called window method and builds feature windows around each time step t with period δ as described in Equation 3.3. For example, with a period of 14 hours I get a total amount of $(60/15 \cdot 14) \cdot 25 = 1400$ features. The highest outlier scores are obtained by averaging the scores of 30 runs

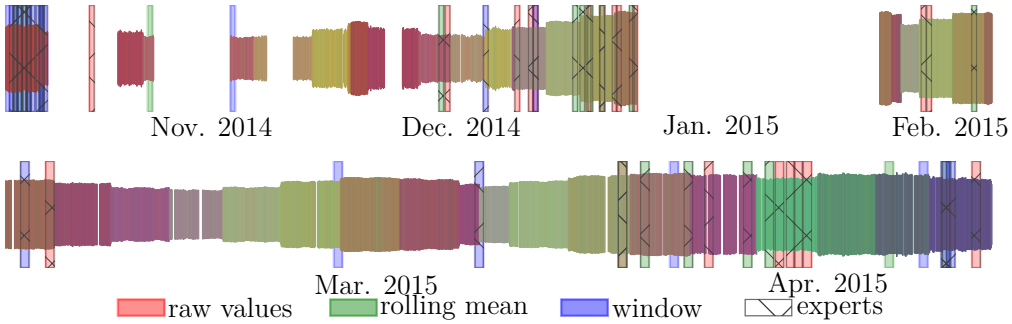


Figure 3.2: Reduced time series to four dimensions and marked top-k-outliers and the experts' evaluation. Every found event by an expert must also be marked by the method, since only results detected by top-k-time are shown to the experts.

with a LOF neighborhood size of 75.

The period parameter δ is set to 14 hours. This corresponds to the tidal cycle of 12 hours and 25 minutes plus a margin to take deviations into account. I choose this period because many extreme events in this time frame are expected by the experts.

3.4.3 Results And Further Improvement

The survey results are shown in Figure 3.2. As I could not show all features at once in one plot, I applied a dimensionality reduction method and reduced the data to four dimensions plus time dimension. The performing method is ISOMAP with 156 neighbors. This nonlinear method utilizes a neighborhood graph to find a low-dimensional embedding for high-dimensional data. I drew inspiration from [80], where three dimensions are mapped to the RGB-values. In addition, I represent the last dimension as the height of the plot. The colored areas are the top-20-time events detected by the three feature methods. If an event is confirmed by at least one expert, it is highlighted with hatches.

The experts confirm 50 of 60 events (83.33%) to be special. They never fully agree on a single event. In seven cases, half of the experts mark the

same events. It was expected that the experts have different priorities in marking events, because of different, but overlapping areas of expertise they are interested in other events. The short text descriptions supports this, for example one participant annotated storms while another focused more on the influence of the tides. I also calculated the Fleiss' kappa [39], which is 0.126 and indicates a slight agreement between the experts [84]. This fits my design, because I want to help a wide range of experts.

Figure 3.3a shows the F_1-score for the three methods. The F_1-score is an evenly weighted precision and recall score, which is defined as:

$$F_1 := 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \ . \tag{3.9}$$

I set the k of the top-k-time method to values in a range between 1 and 20. The score grows with increasing k. This is likely the case, because the test set has a high positive rate and the k value represents the basis for the recall value. First, when k is 20, the highest score is obtained with the raw feature method. But at $k = 15$ this method is overtaken by the windowing method, which stays true with decreasing k. The rolling mean method performs either worse or equal compared to the windowing method. In summary, the raw value method has more false positive for its events with the highest outlier score and show a better recall for small k.

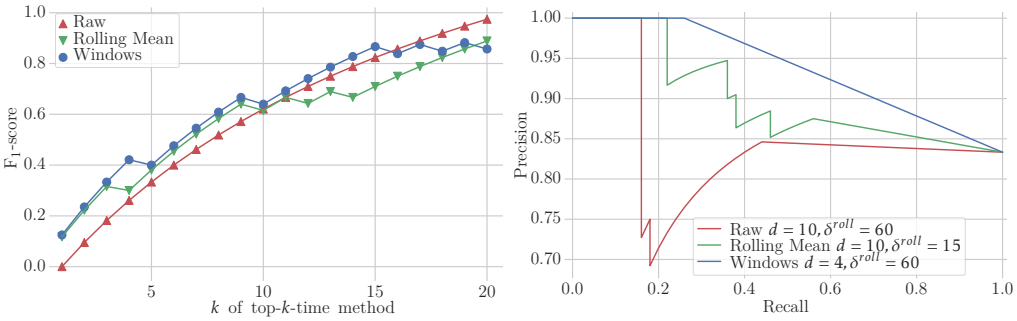
Next, I improve on my initial results with the now labeled data. A tuning of parameters was previously not possible because without labels there was no indication whether the results improve. I change two parameters of the first experimental setup. The first difference is a greater smoothing of the values. Instead of only applying a centered rolling median of 15 minutes as before, now also 60 minutes are used ($\delta^{roll} \in \{15, 60\}$). The dataset size remains the same, because I still take every 15th data point, but the smoothing can be stronger. As second change I employ ISOMAP, the same dimensionality method I used for the visualization in Figure 3.2, to

reduce the features to 4 or 10 features ($d \in \{4, 10\}$). This decreases the total amount of features per pattern for the window method drastically from 1400 to 224 ($= 4 \cdot 56$) or 560 ($= 10 \cdot 56$) for the period of 56 (14 hours with 15 minute sampling).

The results are presented in Table 3.1. I utilize the average precision score, it approximatly represents the area under the precision-recall-curve and takes the ranking of scores into account. To that end, the outlier scores have been normalized with Gaussian normalization as in [81]. All feature methods are able to improve on their previous results with the reduced number of features through ISOMAP. The windowing method initially has the lowest precision score, although it is the only method that considers multiple time steps. This might be due to the curse-of-dimensionality, where distances get practically meaningless in high dimensional space. With fewer features and smoother values, the windowing method performs best, because it avoids this curse and can make use of the additional time steps. In Figure 3.3b, I display the precision-recall-curve of the best methods w.r.t. the average precision score. It shows that the precision never falls below 0.5 precision, which indicates that all methods are better than random predictions. Further, the average precision score results are confirmed, because the windowing method curve is above the other curves and thus better performing.

Table 3.1: Comparison of feature handling methods. The highest scores of each method are highlighted with bold fonts and the *-symbol shows the highest overall score.

		Average Precision Score					
	$d =$	original		4		10	
Methods	$\delta^{roll} =$	15	60	15	60	15	60
Raw Values		0.821	0.803	0.841	0.807	0.761	**0.850**
Roll. Mean		0.855	0.877	0.830	0.885	**0.901**	0.850
Window		0.800	0.801	0.846	**0.938***	0.809	0.801



(a) Change of average F_1-score with increasing k for the feature handling methods.

(b) The precision-recall-curve of the best parameters for the feature methods.

Figure 3.3: Results of expert evaluation of the feature methods for LOF.

3.5 Conclusion

This chapter showed an extreme event detection task for time series data on a part of the TSS dataset. I employed the unsupervised LOF algorithm that produces an outlier score with three different methods to handle features. To choose the extreme events, I extended the top-k-ranking approach to be applicable for time series data. Afterwards, a group of experts examined 60 of the most likely events found by my approach. The results reveal a precision rate of 83.33%. I increased the average precision score by employing ISOMAP as dimensionality reduction method. A novel visualization was introduced that can display four features from the reduced high-dimensional feature space of the time series. This visualization potentially can aid in detecting extreme events for long time segments.

4

Imputation with Penalized DTWkNN Ensembles

4.1 Introduction

In the last chapter, I searched a part of the TSS dataset for special events, but 29.6% of the data were missing. I filled part of these missing data with a standard imputation algorithm. Although the results were satisfactory, I noticed that mismatched events had qualitative feedback suggesting that the afflicted series are behaving unnaturally, but are not special events. This unnatural behavior turned out to be poorly imputed data. The imputation was necessary, because otherwise the algorithm would not be usable, which is also true for many other methods. Reasons for missingness are manifold, for example, a sensor could temporarily fail due to communications issues or irregular maintenance. Simply dropping the afflicted time step from the data would result in information loss, since still available values would be lost and introduce bias. With reasonable imputations, the

bias is smaller and no data needs to be deleted.

Usually, time series data is imputed with classical algorithms, such as linear or spline interpolation [29]. These algorithms work best with smoothed time series, which can introduce more bias into the dataset if this smoothing is too strong. Another disadvantage of these algorithms is that they ignore the multivariability of the data and thereby miss information about feature correlations. The autoregressive integrated moving average (ARIMA) [96] is able to handle multivariability, but is too expensive for many features. To my knowledge, none of these algorithms take into account that values can be missing for multiple consecutive time steps (gaps). The loss in information increases with the length of the gaps.

This chapter introduces a novel imputation algorithm for multivariate time series based on machine learning, called penalized DTWkNN ensembles [106]. I employ the weighted k-nearest neighbors algorithm (kNN) with dynamic time warping (DTW) as distance measure to compare patterns over multiple time steps. Through the DTW comparisons, the resulting model can utilize the auto- and cross-correlations of the time series data. Moreover, correlations between available features and missing features are used to weight the distances. Further, weighting is applied by penalizing longer gaps of missing data with a novel penalty function. I increase the performance by creating an ensemble of DTWkNN models with bagging and other more specialized diversity methods. Empirical evaluation is conducted on part of the TSS dataset, but also on 15 dataset from the UCR time series data mining archive by Keogh and Folias [77]. This evaluation examines several degrees of missing data as well as different minimal missing data intervals. Finally, I apply the method to the combined TSS and BEFmate dataset.

This chapter is organized as follows. First, I discuss the related work in Section 4.2. Next, I create a better formal understanding of missing data and its underlying mechanism in Section 4.3. Section 4.4 presents

existing imputation algorithms such as linear interpolation and kNN. How to compare two time series with DTW is described in Section 4.5. In Section 4.6, I present my DTWkNN ensemble algorithm, which I evaluate experimentally in Section 4.7. The algorithm is then applied to the TSS and BEFmate dataset in Section 4.8. I conclude this chapter in Section 4.9.

This chapter is partly based on the following published paper:

> Oehmcke, S., Zielinski, O., and Kramer, O. (2016). kNN ensembles with penalized DTW for multivariate time series imputation. In *International Joint Conference on Neural Networks (IJCNN)*, pages 2774–2781. IEEE.

4.2 Related Work

Most imputation methods based on machine learning are created for applications without time correlations [45, 71, 74]. In the following, I will give examples for imputation with machine learning and evaluate their use of multivariate imputation as well as their consideration of gaps.

The kNN algorithm established itself as a robust choice for imputation. It has shown superior performance compared to algorithms such as CN2 and C4.5 [11]. Another example with kNN is provided by Troyanskaya *et al.* [141] in biology, applied to incomplete DNA data. There, kNN delivered the best results compared to single value decomposition and row average imputation. Further, Hsu *et al.* [62] have also used DNA data and kNN, but utilized DTW as distance metric instead of Euclidean distance. Their model has been trained on a complete dataset. They have employed different variants of DTW and compared their predictive and computational performance. Since DNA data is univariate, no multivariate testing has occurred. Gaps in the data were also not taken into account.

Rahman *et al.* [118] have proposed a method to deal with lagged correlations in multivariate time series data. To that end, they have extended

kNN to weight features depending on their cross-correlation with missing values. They have also applied a Fourier transformation as imputation method and have combined the outputs. Although the author did not design their experiments around consecutively missing values, they state that the Fourier transformation imputation suffers more from longer gaps than kNN.

EMDI, an ensemble approach by Pan *et al.* [110] has been applied to the application of epistatic miniarray profiling. Here, multiple imputations are combined to a single ensemble output with local and global algorithms, including kNN, local least squares imputation, and matrix completion. The ensemble takes into account the diversity of the ensemble members by weighting the ensemble output accordingly. Their experiments have only considered univariate data and ignored the effects of consecutively missing values. As a side note, the idea of using multiple imputation to get an overall more accurate result has first been introduced to imputation by Donald Rubin [124]. He has proposed it in the classical context of nonresponse in surveys.

Another imputation application is the online reconstruction of missing data, where Alippi *et al.* [2] have utilized multiple linear and non-linear algorithms. When creating the algorithms, they have paid attention to temporal and spatial redundancies of the features. Further, they have simplified their models by employing dependency graphs to select only relevant features. Only single missing values had been considered, since their work focuses on online learning datasets.

In summary, the kNN algorithm proved to be a reliable choice in many applications and is robust to higher rates of missingness. Although kNN is capable to impute multiple values at once, the related work did not explore this ability. Also, the effect of consecutively missing values has not been explored either.

4.3 Missing Data Theory

Although the imputation of multivariate missing data is related to the general regression problem, there are specifics that need to be explained.

4.3.1 Multivariate Time Series Imputation Problem

There are survey imputation definitions by Schafer [127] that I adapted to a multivariate time series imputation definition. The complete time series dataset is an $n \times d$ matrix $X := (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$. A single input is a vector $\boldsymbol{x}_t := (x_1, \ldots, x_d)^T \in \mathbb{R}^d$ at some time $t \in \{1, \ldots, n\}$. In addition, there is a missing indicator matrix M with the same shape as X and a single missing vector $\boldsymbol{m}_t := (m_1, \ldots, m_d)^T \in \{0, 1\}^d$. The entries of this missing indicator matrix are either 1 or 0 to signalize a missing or existing value.

To separate the missing data X^{miss} and the existing data X^{obs}, the indexing operator $\ominus$ is introduced:

$$\boldsymbol{x}_t \ominus \boldsymbol{m}_t := \left(x_j \mid \forall j \in \{1, \ldots, d\} : \begin{cases} x_j & \text{if } m_j = 1 \\ - & \text{else} \end{cases} \right) \tag{4.1}$$

Every feature of an input vector $\boldsymbol{x}_t$ that is marked with 1 in $\boldsymbol{m}_t$ is extracted. For matrices. this operator works element-wise. The observed and missing matrices are then defined as:

$$\begin{aligned} X^{obs} &:= X \ominus \neg M &&:= \left(\boldsymbol{x}_1 \ominus \neg \boldsymbol{m}_1, \ldots, \boldsymbol{x}_n \ominus \neg \boldsymbol{m}_n \right) \\ X^{mis} &:= X \ominus M &&:= \left(\boldsymbol{x}_1 \ominus \boldsymbol{m}_1, \ldots, \boldsymbol{x}_n \ominus \boldsymbol{m}_n \right). \end{aligned} \tag{4.2}$$

For example, let us assume we have four 2-dimensional patterns ($d =$

$2, n = 4$) and a missing indicator matrix M:

$$X = \begin{pmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \end{pmatrix}, \; M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{aligned} X^{obs} &= \begin{pmatrix} 1 & - & - & - \\ - & 4 & 6 & - \end{pmatrix} \\ X^{miss} &= \begin{pmatrix} - & 3 & 5 & 7 \\ 2 & - & - & 8 \end{pmatrix} \end{aligned} \tag{4.3}$$

As in regression, we have a model f for the input $\boldsymbol{x}_t$ that aims to approximate the target vector $\boldsymbol{y}_t$ by minimizing a loss function L. But the input vector $\boldsymbol{x}_t$ is coming from the complete observations X^{obs} and the target vector $\boldsymbol{y}_t$ is part of the missing observations X^{miss}. Each unique combination of the missing indicator vector $\boldsymbol{m}$ theoretically creates a new regression problem. This can reduce the available data for model training significantly. In our example, four unique problems are possible, although $\boldsymbol{m} = (1, 1)^T$ is not represented as learning with no information is unlikely. Further, the first pattern can only employ the fourth as training set and the second pattern is able to utilize the third and the fourth, because of compatible missing masks. In practice, not every combination is present. When testing an algorithm, the missing observations are available to calculate the error measure. If this is not the case, one needs to look at other quality measurements, such as the stability of variances and covariances.

4.3.2 Reasons for Missing Values

There are various causes why a value may be missing and often one cannot know the exact reason [29, 45]. Nevertheless, those causes can be defined as the probability that the missing indicator matrix M takes certain values depending on the observed matrix X^{obs} and the missing matrix X^{mis} as well as an independent set of parameters ξ. If the lack of values only depends

on the dataset, they are Not Missing At Random:

$$\text{NMAR} := \Pr(M|X^{obs}, X^{mis}, \xi) = \Pr(M|X^{obs}, X^{mis}, \xi) \; . \tag{4.4}$$

In this case, it is possible to account for the missing values by creating a model for the assumed missingness process. For example, the TSS would not measure water conductivity at $7.5\,\mathrm{m}$ if the water depth measured only to be $6\,\mathrm{m}$ because the conductivity sensor is not in the water. When the reasons only depend on the independent set of parameters, the values are Missing Completely At Random:

$$\text{MCAR} := \Pr(M|X^{obs}, X^{mis}, \xi) = \Pr(M|\xi) \; . \tag{4.5}$$

This would be the case with random network failures that are not part of the dataset. Finally, if the missing indicator matrix depends on the complete set, then values are Missing At Random:

$$\text{MAR} := \Pr(M|X^{obs}, X^{mis}, \xi) = \Pr(M|X^{obs}, \xi) \; . \tag{4.6}$$

For instance, a water conductivity measurement is often missing after a certain value threshold of the water temperature measurement has been exceeded, but the reason for this is a hardware fault rather than the observed environmental system. In practice, most imputation algorithms assume MCAR or MAR because the causes for missingness are often unknown.

Throughout this work, I assume the MCAR mechanism because the marine application mostly suffers from random failures and network issues. Of course other NMAR reasons for missing values exist, but are sparse and not feasible to account for.

Usually, the parameter set ξ only contains the degree of missing data, but for the time series task, it is also important to regard the minimal

interval of missing data. The degree of missing data $r_d \in [0,1]$ gives a ratio of missing values. With this, the number of missing values approximates to: $r_d \cdot n \cdot d$. To gain more control over the number and length of consecutively missing values, short gaps, the minimal interval length of missing data $r_i \in \mathbb{N}$ is introduced. This is important because otherwise there will be long gaps with a high degree of missing data r_d. This would not be a realistic scenario, as sensor applications are almost always missing for more than one time step.

4.4 Imputation Algorithms

Imputations is not a new problem and there exist many standard algorithms [29]. For instance, the average fill is often applied, it replaces missing values with their average values. Depending on how large the standard deviation of the dataset is, this potentially introduces a bias. Linear interpolation or quadratic interpolation are repairing the gaps at time t by defining a function between two known points $(\boldsymbol{x}_{t'}, \boldsymbol{x}_{t''})$ with $t' < t < t''$. For linear interpolation it is:

$$\boldsymbol{x}_t := \frac{\boldsymbol{x}_{t'}(t'' - t) + \boldsymbol{x}_{t''}(t - t')}{t'' - t'} \ . \tag{4.7}$$

These interpolation algorithms are easy to apply and fast to compute. They perform well for small gaps in the data, but are inaccurate for larger gaps. Another downside to the before mentioned algorithms is that they only work in a univariate way. This means, they are unable to account for multiple features of the dataset at once.

A machine learning algorithm for regression problems can be adapted to the value imputation problem. This is done by training a new model for every unique combination of missing features [45]. To compare my novel imputation method, I use Random Forest [19] as state-of-the-art ensemble algorithm that shows good out-of-the-box performance.

4.4.1 k-Nearest Neighbors

The kNN algorithm is a simple but effective method that is successful in applications such as microarray gene expression analysis and outcome prediction [111] or fault detection [59]. It relies on the principle that the target output $f_{\text{kNN}}(\boldsymbol{x}_t, \boldsymbol{m}_t, j) := \hat{y}_t$ for an unseen input vector $\boldsymbol{x}_t$ is the average target of the k-closest input vectors from the training set [45, 57]. In the missing data task, the target y will be a missing feature value x_j of the input with $j \in \{1, \ldots, d\}$. The training set consists of complete inputs that have the missing feature, so the missing indicator vector $\boldsymbol{m}$ must be 0 at the j-index and also at all feature positions were the input vector has values. First, the k-nearest neighbors in regard to a distance measure Δ are located with a neighbor function $\mathcal{N}_{\text{k}}(\cdot)$:

$$\begin{aligned} \mathcal{N}_{\text{k}}(\boldsymbol{x}_t, \boldsymbol{m}_t, j) := k\text{-}\underset{x_{t'} \in L}{\operatorname{argmin}} \Delta(\boldsymbol{x}_t \ominus \neg \boldsymbol{m}_t, \boldsymbol{x}_{t'} \ominus \neg \boldsymbol{m}_t) \\ \text{with } L := \{\boldsymbol{x}_{t^*} \mid \forall t^* \in \{1, \ldots, n\} : (\boldsymbol{m}_{t^*} \rightarrow \boldsymbol{m}_t) \wedge \Big(M_{(t^*,j)} = 0\Big)\} \ . \end{aligned} \tag{4.8}$$

Set L contains all compatible missing masks that can be utilized as training set. These compatible masks must have the same available features, but can also have more, although these features cannot be used for neighborhood comparisons. Then, the values of these k close-by targets are aggregated:

$$f_{\text{kNN}}(\boldsymbol{x}_t, \boldsymbol{m}_t, j) := \frac{1}{\text{k}} \sum_{\boldsymbol{x}_{t'} \in \mathcal{N}_{\text{k}}(\boldsymbol{x}_t, \boldsymbol{m}_t, j)} (\boldsymbol{x}_{t'} \ominus \neg \boldsymbol{m}_t)_j \ . \tag{4.9}$$

A special role takes the distance measure Δ, because it can change the outcome of the prediction completely. Euclidean distance is employed as distance measure Δ most of the time (see Equation 3.6). For imputation, this is disadvantageous as only patterns with the same missing vectors can be utilized. Moreover, Euclidean distance ignores the sequential properties

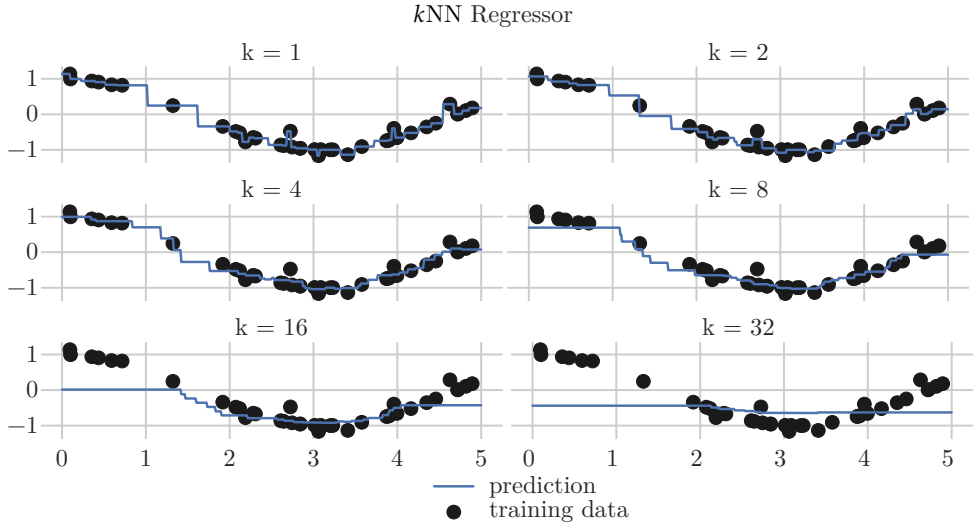


Figure 4.1: Difference in predictions with changing number of neighbors k. Time is on the x-axis and serves as input values. The target output is on the y-axis.

as it only compares one pattern at a time.

The next important parameter of kNN is the number of neighbors k. Figure 4.1 shows the influence of k in a simple regression example that aims to approximate a cosine function with some noise in the training data. On the one hand, if k is too small, it overfits to the closest pattern and cannot account for noise in the data. On the other hand, when k is to large, the prediction overgeneralizes and the average of the training set is returned.

4.5 DYNAMIC TIME WARPING

As mentioned in Section 4.4.1, standard distance measurements are usually not well suited for time series comparisons. The idea of the DTW distance measure is to align two series and return the distance of these series. A flexible alignment is found by creating a "warping" path between sequences. Figure 4.2 shows an exemplary alignment. Introduced for a

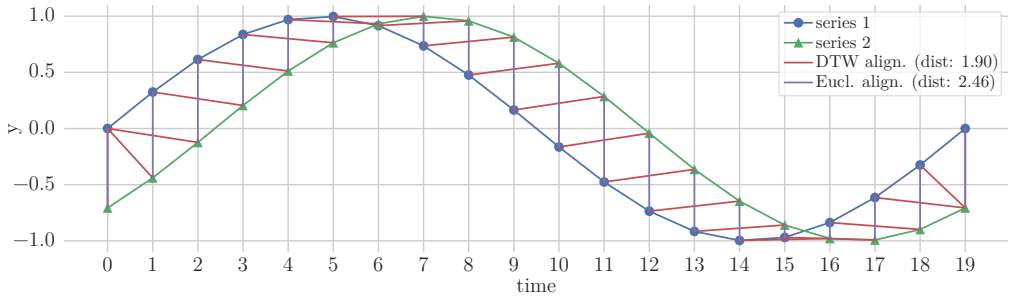


Figure 4.2: Exemplary DTW alignment of two temperature time series compared to Euclidean distance.

speech recognition task [83], DTW was also applied in other data mining tasks, such as gesture recognition [137] or time series databank tasks [32].

DTW aligns two sequence patterns:

$$X_{t-\delta+1:t} := (\boldsymbol{x}_{t-\delta+1}, \ldots, \boldsymbol{x}_t)$$

and

$$X_{t'-\delta'+1:t'} := (\boldsymbol{x}_{t'-\delta'+1}, \ldots, \boldsymbol{x}_{t'})$$

at different times t, t' and sequence lengths δ, δ'. The calculation for the minimal path distance is depicted in Algorithm 1. At first, the entries of the path matrix D are set to infinity in Line 1. The exception is the very first entry that is set to zero as a starting point in Line 2. Iteration after iteration, the matrix D is filled with the sums of the minimal path distances. This is achieved by taking the sum of the previous minimal path distance p as seen in Line 6 and add it to the current distance s. Finally, the result of the DTW calculation is the minimal complete path distance at $D_{\delta,\delta'}$. The pointwise distance measurement Δ usually is the Euclidean distance. In contrast to using the Euclidean distance for the complete comparison, the sequence widths δ and δ' can have different sizes. While the runtime is usually higher than the Euclidean distance, it is still feasible. With lower bounding, the runtime is closer to $O(\delta)$ than to $O(\delta^2)$ [119].

Algorithm 1: DTW-algorithm [83]

input : input series X, time t, sequence widths δ, time t', sequence width δ', distance metric Δ

$D \leftarrow$ create $(\delta+1)\times(\delta'+1)$ matrix full of ∞
$D_{(0,0)} \leftarrow 0$
// Iterate through both series
for $l \leftarrow 0$ **to** δ **do**
    for $j \leftarrow 0$ **to** δ' **do**
        // calculate distance and minimal path
        $s \leftarrow \Delta(\boldsymbol{x}_{(t-\delta+1)+l}, \boldsymbol{x}_{(t'-\delta'+1)+j})$
        $p \leftarrow \min(D_{l,j}, D_{(+1,j}, D_{l,j+1})$
        $D_{(l+1,j+1)} \leftarrow s+p$
return $D_{(\delta,\delta')}$

4.6 Penalized DTWkNN Ensemble

The method proposed here, called penalized DTWkNN, combines a simple linear interpolation imputation with kNN that applies a normalized and weighted DTW as its distance measure Δ. Moreover, ensemble techniques are employed, such as bagging or varying the penalty strength. This increases diversity and results in improved performance.

4.6.1 Preprocessing: Windowing and Linear Interpolation

Because I am dealing with time series, it is important to consider more than one step for the input of my machine learning algorithm. One input series $X_{t-\delta+1:t} := (\boldsymbol{x}_{t-\delta+1}, \ldots, \boldsymbol{x}_t)$ is built for at time step $t \in (1, \ldots, n)$ with a certain sequence width δ. This is repeated for every time step to create the dataset used by DTWkNN. For univariate data, meaningful DTW comparisons are possible, even when there is missing data, as seen in [62]. This is partly possible because DTW is able to compare series with different widths.

In the case of multivariate time series, the missing values for DTW are problematic, especially when different features have missing values at different time steps. Let us assume some input $X_{t-\delta+1:t}$ and its missing indicator matrix:

$$M_{t-\delta+1:t} = \begin{matrix} \\ x_1 \\ x_2 \end{matrix} \begin{matrix} \begin{matrix} t-3 & t-2 & t-1 & t \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{matrix},$$

with width $\delta = 4$ and dimensionality $d = 2$ at time t. If I delete the data column-wise, only one time step remains, because only at $t-2$ both features are present. In the worst case scenario, when $t-2$ would contain a missing value, the entire input would be unusable. If I apply individual DTW at every dimension, the returned distance is biased, because x_1 is missing two values and x_2 only one. Another solution would be to impute the missing values with a computationally inexpensive method. This method is linear interpolation, as it works well enough for small gaps and shows better results than spline or Fourier interpolations [147]. The prior imputation guarantees that the sequence width δ is always the same, which potentially speeds up the DTW calculations [119].

The now complete dataset reduces the number of regression problems from one problem for each possible combination ($2^{d \cdot \delta}$) of the missing indicator vector to the number of features d. As a note, the training vector of should always have non-imputed values at time t for the sought after feature. Without this restriction, the model would easily overfit to the used prior imputation.

4.6.2 Multivariate DTWkNN-Distance

There are two basic ways to apply DTW to multivariate time series [131]. The first one finds a minimal path in multidimensional space and thus each dimension depend on the other. The second way, calculates the individual

DTW distance of each dimension and combines them as:

$$\sqrt{\sum_{j=1}^{d}\left(\mathrm{DTW}\left(X_{t-\delta+1:t,j}, X_{t'-\delta+1:t',j}\right)\right)^2}\,. \tag{4.10}$$

In this application, I use this independent DTW calculation because it enables the easy use of different weights for each dimension and not all features are mutable depending on one another. Although some multivariate information gets lost, it is later reintroduced with the correlation matrix weighting and results from [131] suggest more constistent error rates with indiependent DTWThis variant also requires less time to compute results, because the cost to find paths for d sequences with width δ^2 instead of one sequence with width $\delta^{2 \cdot d}$ is increasing slower. Assuming $d = 14$ and $\delta = 10$, the independent DTW needs 1400 pointwise comparisons instead of $10\mathrm{e}28$ for the dependent DTW.

4.6.3 Normalized DTW-Distance

The scaling of features is often not uniform. If kNN is applied on data that has a different scaling, a feature with greater values translates into a greater importance of this feature although it might not be important at all. This is comprehensible for an example from the TSS. Wind direction measurements scale between 0 and 360 degree, while the values for water temperature range from 0 to 15 degree Celsius. A change of wind direction by 20 degree would have a larger impact on the distance measure than a temperature change of 7 degrees.

Normalization solves this inequality by dividing the DTW-distance of a feature by the standard deviation of this feature. The required standard deviation vector $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_d)$ is calculated beforehand from the training

set. Equation 4.10 is redefined to:

$$\sqrt{\sum_{j=1}^{d}\left(\frac{\mathrm{DTW}\left(X_{t-\delta+1:t,j}, X_{t'-\delta+1:t',j}\right)}{\sigma_j}\right)^2}\,. \tag{4.11}$$

If no deviation is calculable, the value for σ is set to 1. This prevents a division by zero error. The centering of values by subtracting the mean is not necessary here, because the distance measure already does this implicitly.

4.6.4 Weighting of Multivariate Outputs with Correlation

Although the normalization of features helps to reduce unfair importance of weights, putting more weight to relevant features is even more effective. These relevant features w.r.t. the missing feature are determined with the Pearson product-moment correlation coefficient [113] because a high correlation indicates a strong relation between two features. A practical example is the high correlation between water temperature and pressure. This can be explained by the faster movement of molecules in hotter water, which results in higher pressure. The use here is inspired by Rahman *et al.* [118], who state that utilizing correlations is good practice if one assumes the MAR or MCAR process.

The Pearson correlation coefficient matrix $C_{l,j}$ describes the linear correlation between feature l and j. It is symmetric and scales between zero and one. The matrix can be applied to the normalized distance measurement because of the independent DTW calculation per feature. A new parameter j' needs to be added to represent the current target feature for the weight matrix W. The final form of the redefined Equation 4.11 is:

$$\Sigma DTW(t,t',j') := \sqrt{\sum_{j=1}^{d}\left(\frac{\mathrm{DTW}\left(X_{t-\delta+1:t,j}, X_{t'-\delta+1:t',j}\right)}{\sigma_j}\cdot W_{t,j,j'}\right)^2}\,. \tag{4.12}$$

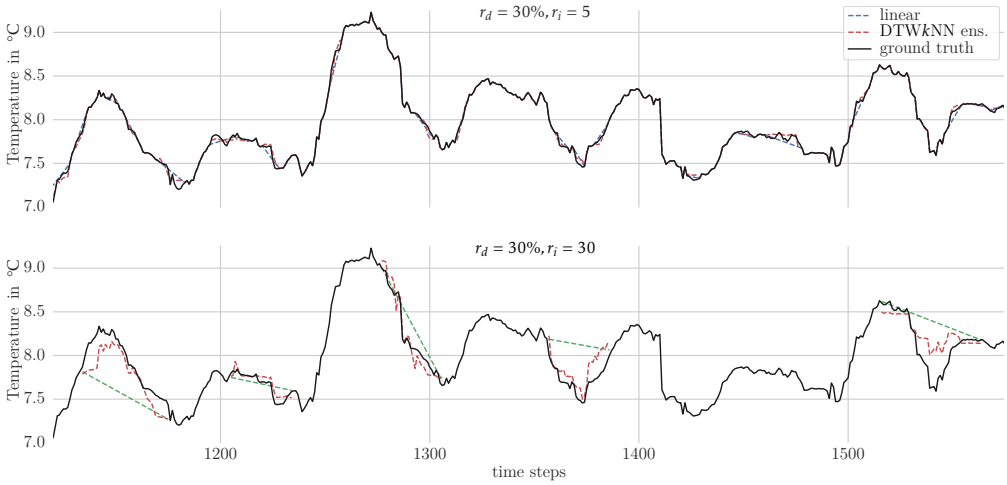


Figure 4.3: The top plot shows performance of linear interpolation and DTWkNN with small gaps ($r_i = 5$), while the bottom plot show larger gaps ($r_i = 30$). Note the performance decrease for linear interpolation.

The $t \times d \times d$ weighting matrix W is defined as:

$$W_{t,j,j'} := \text{abs}(C_{j,j'})^{\alpha}. \tag{4.13}$$

As it is unimportant if the correlation is positive or negative and should only scale the individual distances, I take the absolute values of the global correlation coefficient matrix C. This global matrix C is calculated only once from the entire training dataset. To gain more control over the correlation weights, a variable α is introduced. It stretches the weights with higher values. Optimal parameter settings for α are obtainable with hyperparameter search via cross-validation or gridsearch.

4.6.5 Penalizing Missing Interval Interpolation

The prior imputation by linear interpolation introduces a bias to the dataset. This is mostly apparent when larger gaps are present. In Figure 4.3 different minimal gap sizes are used to show the effect of linear

interpolation and the DTWkNN method. When the DTWkNN algorithm would use these linear interpolations as-is, it could match to the wrong neighbors, resulting in imputations with a high error rate.

Algorithm 2: Creating penalty weight matrix P

input : missing indicator matrix M, sequence width δ, number of time steps n, number of features d

```
// invert M: missing= 0 and not-missing= 1
```
1 $P \leftarrow \sim M$
2 **for** $j \leftarrow 1$ **to** d **do**
3 $\quad i \leftarrow 0$ `// gap width i`
4 $\quad$ **for** $t \leftarrow 0$ **to** n **do**
$\quad\quad$ `// increasing i if value is missing`
5 $\quad\quad$ **if** $M_{t,j}$ **then**
6 $\quad\quad\quad i \leftarrow i + 1$
$\quad\quad$ `// apply penalty to gap`
7 $\quad\quad$ **else if** $i > 0$ **then**
8 $\quad\quad\quad P_{t-i,j}, \ldots, P_{t,j} \leftarrow \frac{1}{i}$
9 $\quad\quad\quad i \leftarrow 0$
10 **if** $c > 0$ **then** `// last gap could be open-ended`
11 $\quad P_{n-i,j}, \ldots, P_{n,j} \leftarrow \frac{1}{i}$
`// adapt penalty for each sequence`
12 **for** $t \leftarrow 1$ **to** n **do**
13 $\quad \boldsymbol{p}_t \leftarrow \frac{\sum_{t'=t-\delta+1}^{t} \boldsymbol{p}_{t'}}{\delta}$
14 **return** P

To weaken the negative effect of the prior imputation, DTWkNN penalizes the linear interpolation with increasing gap size. To that end, a penalty weight matrix P for each input vector and each feature in the test dataset is created. The columns contain the penalty vectors $\boldsymbol{p}_t$ with $t \in \{1, \ldots, n\}$. Algorithm 2 outlines the computation of matrix P. To realize an increasing penalty with larger gaps, the algorithm computes the width of the gap and saves it to variable i at Line 6. The penalty is then the

reciprocal value of this gap width i as seen at Line 8. Finally, the penalty weight vectors are scaled by the sequence width δ at Lines 12 and 13.

An update to Equation 4.13 introduces the penalty matrix P to the complete weight matrix:

$$W_{t,j,j'} := |C_{j,j'}|^{\alpha} \cdot P_{t,j}^{\varepsilon} \ . \tag{4.14}$$

I apply the penalty to each input vector $\boldsymbol{x}_t$ individually, while the correlation weight is introduced only feature-wise w.r.t. the current target feature. To control the severity of this penalty, the variable ε is used. This can be achieved globally with a single value or per feature with a vector of length d.

4.6.6 Efficient Ensemble Building

Often, parameter settings are good for one part of the dataset, but less ideal for another. Finding a global setting that fits all problems is impossible due to the no-free-lunch-theorem. Ensembles avoid this problem by pooling the results of l model outputs into one ensemble output $\bar{F}$, e.g. by averaging their outputs:

$$\bar{F}(\boldsymbol{x}) := \frac{1}{l} \sum_{i=1}^{l} f_i(\boldsymbol{x}) \ . \tag{4.15}$$

with single model f from all possible models $f \in F$.

There is no need for one model that solves the learning problem perfectly, but it is better to employ multiple models that are good at different parts of the learning problem. In other words, the l individual models should show a high diversity [31]. The diversity in a DTWkNN ensemble comes from different parameters settings for each individual model. One of these parameters is the adaption of the penalty weight by drawing the value for exponent ε (see Equation 4.14) from a normal distribution for each ensemble member. The other parameter is the neighborhood size k. This

one is drawn from a uniform distribution at random between some start and end value.

The DTWkNN ensemble also applies bootstrap aggregating (bagging). This is a classic method for ensemble diversity by Breiman [18]. Bagging takes a random subset of the training set for every individual model as new training set. To give more weight to random input vector, this is done with replacement. One training subset consists of n_F input vectors: $n_F := s_r \cdot n$, where s_r is the sampling rate and n is the total number of input vectors.

Frequently, the number of calculations and thus the runtime for an ensemble method increases by the number of employed individual models. In contrast, the runtime for an DTWkNN ensemble is only slightly higher than that of a single model. The reason for this is the normalized distance matrix, which is the most expensive part of kNN, is reusable for each model. Its shape is the same for every model: $m \times n \times d$ with m the number of incomplete patterns, n the number of all patterns, and feature number d. First, the matrix is computed, but the correlation and penalty based weighting is not applied. Then, because of bagging, every model is given a different part of this matrix and the models apply their own weights and choose their neighbors accordingly. Although all models need to apply their own weights, this approach is still fast, because matrix multiplication is a highly optimized standard operation. This is another reason why the independent DTW is best suited for DTWkNN because the separation of the individual feature distances ensures a quick multiplication with the weights of the ensemble members.

4.7 Experimental Analysis

In the following, I experimentally compare my penalized DTWkNN algorithm with classical imputation and state-of-the-art machine learning algorithms on 16 datasets.

Table 4.1: Accumulated best R^2 scores for tested interval and missing rate combinations over all datasets.

intervals r_i =	1				5				15				30				45				
mis. rates r_d =	.1	.2	.3	.4	.1	.2	.3	.4	.1	.2	.3	.4	.1	.2	.3	.4	.1	.2	.3	.4	$\sum$
average fill	0	0	0	0	1	1	1	1	1	2	1	2	1	1	1	1	2	2	2	3	23
linear	4	3	3	2	1	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	17
RF	5	3	3	1	5	3	3	1	5	4	2	1	7	4	2	1	8	6	2	1	67
RF-linear	1	2	3	5	3	3	4	5	4	2	1	2	1	2	2	1	1	0	2	3	47
DTWkNN	1	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	3
DTWkNN ens.	5	8	7	7	6	7	7	8	6	8	12	11	6	9	11	13	5	8	10	9	**163**

4.7.1 Data and Design of Experiment

The selected datasets all show different characteristics, but are all multivariate time series dataset. Their number of patterns n range from 867 up to 2500. The dimensionality d per pattern is at least two or at most 14 features. As I foremost want a method to fill the gaps in my marine datasets, I utilize a part of the TSS dataset (see Section 2.2) from 2015-03-26 22:10 to 2015-04-13 23:50. This time frame is chosen because it shows less sensor failures and thus is more complete than other time frames, which is important for evaluation. The remaining dataset are: ballbeam, balloon, dryer2 sensor, foetal ECG, glassfurnace, greatlakes, pgt50 alpha, pgt50 cdc15, pH-data, phone1, robot arm, shuttle, synthetic, water, and winding. They are provided by Keogh and Folias [77] in their UCR time series data mining archive. I apply no smoothing or resampling to the UCR datasets except for the synthetic dataset where I only use every 50th pattern, because the rate of change is so small that even huge gaps pose no challenge.

The datasets are artificially damaged with different settings for the missing rate r_d and minimal missing interval r_i. This allows me to employ the damaged parts as test sets, because I still have the original values. The tested missing rates r_d are: $r_d \in \{.1,.2,.3,.4\}$. Missing rates above that are ignored because the information density for $r_d > .5$ is so low that it would be more appropriate to collect new data. I set the minimal missing interval r_i to: $r_i \in \{1, 5, 15, 30, 45\}$. Depending on the data resolution of a dataset, imputing these gaps can be of varying difficulty. In addition, the smallest dataset loses 5.19% of its data from a gap with 45 time steps, while the biggest dataset only misses 1.8%. For reproducibility reasons, I run 30 repetitions for each combination of missing rate and minimal missing interval. Every repetition also yields another random missing mask w.r.t. r_d, r_i.

My imputation algorithm is tested as single and ensemble predictor. As classical algorithm, I choose linear interpolation because of its wide use.

Table 4.2: Different ranges for the parameter settings of the used imputation models.

Model	Parameter	Value Ranges
Random Forest	num. estimators	200
	max. tree depth	1, 2, 5, 8, ∞
	max. features	sqrt, log2, all
DTWkNN	sequence width δ	1, 5, 10, 15
	penalty exp. ε	.0, .0625, .125, .25, .5, 1
	covar. exp. α	.0, .0625, .125, .25, .5, 1, 2
DTWkNN ens.	num. estimators	50
	kNN's k	unif(1,15)
	penalty std. $\check{\varepsilon}$	.0, .03125, .0625, .125, .25, .5
	sample rate s_r	.125, .25, .5, .75, 1, 1.25, 1.5

Moreover, I want to show that my algorithm performs better than its preprocessing step. The other machine learning algorithm is Random Forest. This algorithm is also an ensemble algorithm and offers good performance without much tuning of hyper-parameters. I also employ Random Forest with linear interpolation as preprocessing step (lin-RF) to verify that the simple combination of linear interpolation with any machine learning algorithm is not sufficient to solve imputation problems. To keep the introduced bias low, I interpolate all values except for the target I am currently imputing. With preliminary studies, I tuned the parameter settings of the tested algorithms. Note that I take the settings from the single DTWkNN for the ensemble algorithm. Values for the parameters are shown in Table 4.2.

The datasets are multivariate and I also consider that multiple values can be missing at the same time. Because of that, I have many imputed values of different scales that need to be evaluated together. Considering every feature individually would be too time-consuming, highly complex, and incomprehensible. That is why I use the R^2 score averaged over all

d features, which ranges from $[-\infty, 1]$:

$$R^2(X, \hat{Y}, \bar{Y}) := \frac{1}{d} \sum_{j=1}^{d} \left(1 - \frac{\sum_{i=1}^{n} \left(\hat{Y}_{i,j} - f\left(X_{i,j}\right) \right)^2}{\sum_{i=1}^{n} \left(\hat{Y}_{i,j} - \bar{Y}_{i,j} \right)^2} \right), \tag{4.16}$$

with the true output $\hat{Y}$, the predicted output vector $f(X)$, and the arithmetic mean $\bar{Y}$ of the test dataset. Higher values indicate better models, although a value of 1 could imply the presence of overfitting problems. If the R^2 score is less than or equal to zero, the imputed value is equal or worse than inserting the mean value of that dataset.

4.7.2 Results of Experiment

A comprehensive overview of the R^2 score performance for each method is given in Table 4.1. Each column represents another r_d, r_i combination for every dataset. Whenever a method has the highest mean score minus the standard deviation, it receives a point. In total, there are 320 combinations that can give a point, resulting from four missing rates, five missing intervals, and the 16 datasets. I also considered another standard imputation method here, the fill with average values. Although its overall performance is not very good, it beats the other methods for longer intervals in 23 cases. Filling the missing values from linear interpolation yields good results for short missing intervals, but as expected, this method does not work for longer intervals. In contrast, Random Forest is able to deal with long gaps, but the performance suffers with increasing missing rates. With the linear preprocessing step for Random Forest, it is able to outperform linear interpolation, but the bias for longer periods is also present. Eventually, my DTWkNN ensemble achieved the best results in around 51% of the cases. Four times the single DTWkNN model is best, but in general the experiments confirm that the ensemble method works better in most scenarios.

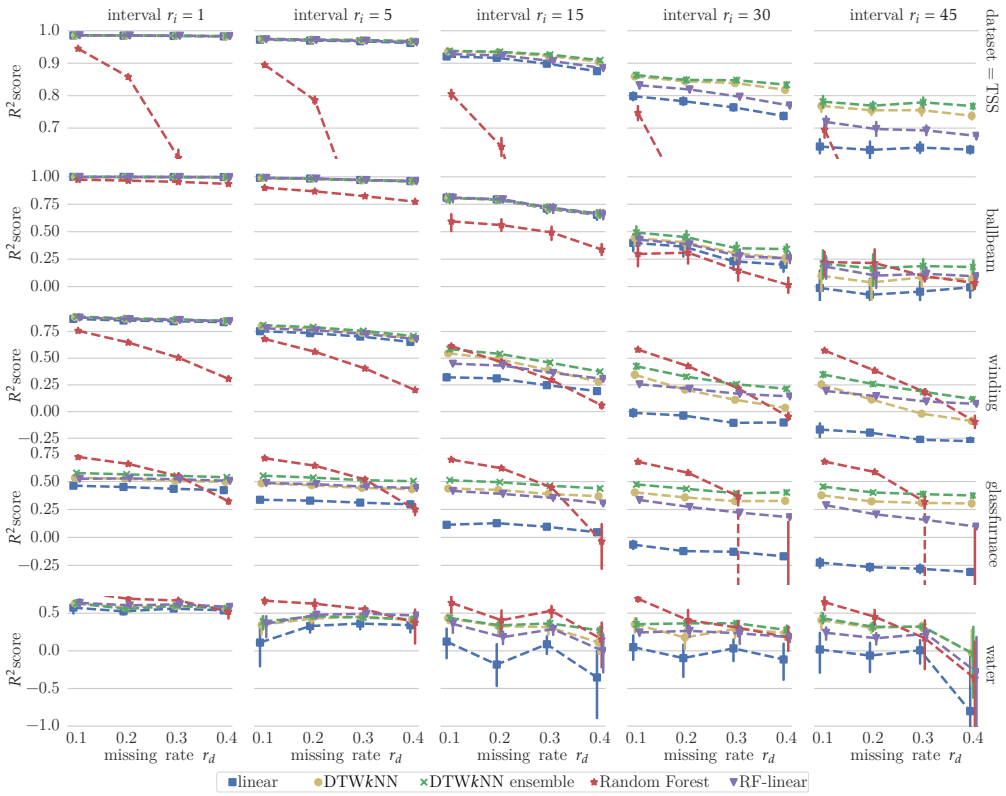


Figure 4.4: Performance on TSS, ballbeam, winding, glassfurnace, and water dataset with different interval settings r_i and missing rates r_d. To improve visibility, results from different algorithms are slightly shifted on the x-axis.

In Figure 4.4, I display the concrete R^2 scores on the TSS, ballbeam, winding, glassfurnace, and water datasets. An important finding is that all imputation methods show similar results for small minimal intervals, but differ from each other with longer intervals. The exception is Random Forest without linear interpolation, although often good at small missing rates, the performance always degenerates with greater missing rates. This is a result of under-fitting as the training set of Random Forest is limited by compatible missing masks. With linear interpolation as preprocessing step, Random Forest proves to be superior to linear interpolation, but of-

ten cannot surpass the other models. The imputations of single DTWkNN are mostly stable over the missing rates. Although the increase in gap length hurts the DTWkNN performance, the impact is less apparent than in the other methods. In the water dataset, I observe that the highest missing rate paired with the highest minimal interval length can diminish the performance under zero for all methods.

Table 4.3 presents detailed scores on the datasets balloon, foetal ECG, greatlakes, and TSS. On the balloon dataset, Random Forest outperforms the other algorithms, probably due to the small number of features and the low time resolution. This does not allow the other algorithms to benefit from a multivariate data structure or the time context. For the foetal ECG and greatlakes datasets, Random Forest again starts with good imputations, but under-fits with higher missing rates. The DTWkNN ensemble does not have this problem. An interval length of 15 and 45 corresponds to 2.5 and 7.5 hours on the TSS dataset, but my ensemble method provides imputations of good quality.

4.8 Penalized DTWkNN Applied on the Combined TSS and BEFmate Dataset

After I validated the potential of the penalized DTWkNN method, I applied it on the combined TSS and BEFmate dataset. This is necessary because 13% of the datasets' values are missing. Further, when combined with the target flow sensor, this number would increase to $\sim 55\%$ missing data. Such a big loss in information would be fatal for any further analysis, which would multiply, when we consider more than one time step at once. If standard methods are employed, the introduced bias is too large and if no imputation method is applied, under-fitting could occur. My penalized DTWkNN method is better suited for this task.

Table 4.3: R^2 scores of four datasets for interval lengths of 15 and 45. Highlighted in bold are the highest scores of a dataset for each combination of missing rate r_d and interval r_i.

		r_i =	15				45			
Dataset	Imputation Model	r_d =	0.1	0.2	0.3	0.4	0.1	0.2	0.3	0.4
balloon	Random Forest		**0.90**	**0.80**	**0.69**	**0.60**	**0.88**	**0.75**	**0.67**	**0.53**
	RF-linear		0.33	0.27	0.20	0.13	0.16	0.19	0.11	0.09
	DTW*k*NN ensemble		0.37	0.32	0.26	0.12	0.32	0.28	0.21	0.11
	linear		−0.17	−0.29	−0.26	−0.41	−0.42	−0.35	−0.35	−0.47
foetal ECG	Random Forest		**0.91**	**0.87**	0.70	0.29	**0.89**	**0.81**	0.55	−0.19
	RF-linear		0.45	0.34	0.28	0.21	0.26	0.24	0.18	0.10
	DTW*k*NN ensemble		0.82	0.76	**0.72**	**0.62**	0.72	0.67	**0.59**	**0.62**
	linear		−0.13	−0.17	−0.22	−0.27	−0.80	−0.62	−0.63	−0.61
greatlakes	Random Forest		**0.77**	**0.75**	**0.68**	0.56	**0.73**	**0.67**	0.55	0.34
	RF-linear		0.66	0.66	0.64	0.61	0.49	0.51	0.46	0.43
	DTW*k*NN ensemble		0.72	0.71	**0.68**	**0.65**	0.64	0.62	**0.58**	**0.52**
	linear		0.51	0.54	0.52	0.50	0.14	0.25	0.24	0.23
TSS	Random Forest		0.81	0.64	0.11	−1.00	0.70	0.35	−0.59	< -1
	RF-linear		0.93	0.92	0.91	0.88	0.72	0.70	0.69	0.68
	DTW*k*NN ensemble		**0.94**	**0.94**	**0.93**	**0.91**	**0.78**	**0.77**	**0.78**	**0.77**
	linear		0.92	0.92	0.90	0.88	0.64	0.63	0.64	0.63

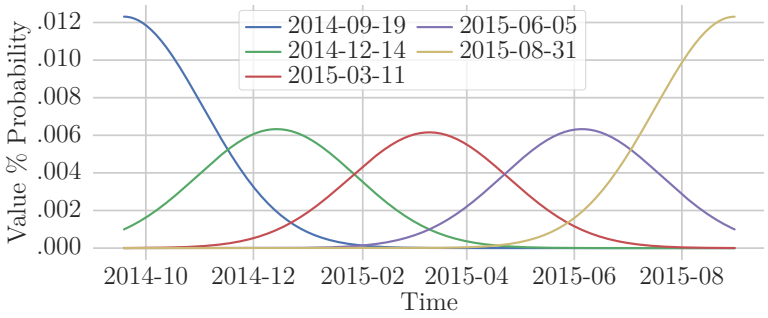


Figure 4.5: Probability based sampling for five example dates for the employed period.

I made practical changes to the algorithm. First, the computational time is decreased by randomly drawing the training data for a single missing value from a normal distribution with 1.5 months as standard deviation. In Figure 4.5 the probabilities for choosing a training pattern are visualized for five example dates. Then, I employ the outlier-robust Spearman correlation instead of Pearson correlation for the global weighting of features. Another advantage of the Spearman correlation is that it does not necessarily assumes Gaussian data. Instead of a sampling rate s_r I use a particular amount of patterns $s_\#$ for each ensemble member.

As there are no real values for the missing gaps, the search for good parameters cannot rely on the R^2 score as it has in the previous section. Therefore, I have decided to compare the change in the Spearman correlation values between the imputed set X^{imp} and the incomplete dataset X^{obs}:

$$\text{corr score} := \sqrt{\sum_{d'=1}^{d} \sum_{d''=1}^{d} (\text{corr}(X_{:,d'}, X_{:,d''}) - \text{corr}(X^{obs}_{:,d'}, X^{obs}_{:,d''}))^2} \ . \tag{4.17}$$

This follows a suggestion by Garcia *et al.* [45], although they did not go into detail about the direct correlation comparison. Preliminary tests showed that the correlation α from Equation 4.14 is best set to one. For each run, the same 1000 random missing samples are imputed by each method. I decided to test the following parameter settings with 25 runs:

Table 4.4: Top and worst five parameter settings of penalized DTWkNN for the combined dataset.

penalized DTWkNN ensembles rank	δ	kNNs k	$\bar{\varepsilon}$	$\check{\varepsilon}$	$s_{\#}$	corr score
1	7	(1, 5)	.500	.050	5000	6.577
2	3	(1, 5)	.500	.050	5000	6.580
3	7	(1, 10)	.500	.050	5000	6.605
4	3	(1, 10)	.500	.050	5000	6.608
5	7	(1, 5)	.500	.100	5000	6.613
⋮	⋮	⋮	⋮	⋮	⋮	⋮
-5	7	(1, 10)	.125	.050	2500	8.013
-4	9	(1, 10)	.125	.025	2500	8.016
-3	7	(1, 10)	.125	.025	2500	8.151
-2	5	(1, 10)	.125	.050	2500	8.218
-1	5	(1, 10)	.125	.025	2500	8.307
standard methods						
Forward and Backward Filling						6.892
Linear Interpolation						6.921
Cubic Interpolation						8.785

- DTW sequence width: $\delta \in \{3, 5, 7, 9, 13\}$
- range of neighbors: $k \in \{(1, 10), (1, 5)\}$
- number of bagging samples: $s_{\#} \in \{1000, 2500, 5000, 7500\}$
- penalty mean: $\bar{\varepsilon} \in \{.125, .25, .5, .062\}$
- penalty standard deviation: $\check{\varepsilon} \in \{.025, .05, .1, .012\}$

Table 4.4 shows the results. The top five results are better in maintaining the correlations than the standard methods from the literature. I use the best setting to impute the complete dataset, which is then employed in Chapter 6 and 7.

4.9 Conclusion

In this chapter, I described and evaluated the penalized DTWkNN ensemble, which is a novel imputation algorithm that is able to reconstruct data with large gaps. This algorithm is designed to work with multivariate time series data, such as the TSS and BEFmate dataset. It utilizes standard linear interpolation as its first step and then weighs the distance matrix of kNN according to the correlation to the missing value and the length of the gap. The DTW distance measure enables a reliable comparison of time series and ensemble techniques further improve results while relying less on exact parameter settings.

In an experiment on 16 different time series, my algorithm outperforms standard as well as state-of-the-art methods in 51% of all cases. Interestingly, other algorithms, such as Random Forest under-fitted at higher missing rates, since the training set is getting too small. The bias from linear interpolation preprocessing increased with wider minimal missing intervals, but my penalty weighting counteracts this and offers reliable results. One requirement for the algorithm is that it can be applied on highly correlated time series data, so it can compensate missing features with correlated ones. This is also a limitation, if the missing feature only correlates with features that are also missing, then my weighted distance matching cannot offer any insightful information for the imputation.

In a final experiment, I applied the new method on the combined TSS and BEFmate dataset. To measure performance, I tested if correlations are kept intact and thus found optimal parameters for the algorithm. Now, a complete dataset is available for modeling a virtual broken sensor with deep learning methods.

Part II

Virtual Sensors with Deep Learning

In the previous part, I introduced the TSS and BEFmate dataset. I also provided a novel imputation algorithm to fill the gaps in my data in the last chapter. These are the necessary requirements to learn expressive machine learning models on these data.

My goal in this part is to create a virtual sensor, which in my application is a regression problem with time series aspects. This prediction task is not a forecast into the future, since the input data is measured at the same time as the target output. Another difference to forecast models is that previous time steps of the target sensors are not part of the input vector, only nearby sensors are employed as inputs.

Just as electricity transformed almost everything 100 years ago, today I actually have a hard time thinking of an industry that I don't think AI will transform in the next several years.

Andrew Ng

5 Deep Learning Foundations

5.1 Introduction

This chapter introduces the fundamentals of deep learning. Deep learning is the use of many hidden layers in a neural network, although there is no clear definition when a network is *deep*. It showed recent success in various application, such as image tasks [49, 58, 82, 102], audio processing [56, 64, 149], wind power prediction [28, 30, 146], language processing/modeling [27, 52, 135], and many more. This is due to extended or new algorithms as well as more capable hardware, e.g. parallel computations on the graphics processing unit (GPU). Moreover, other machine learning approaches are either not feasible to train on big datasets, because they are computationally too expensive (e.g. kNN), or they rely on constraining data distribution assumptions (e.g. Gaussian Mixture Models).

The reason why I create the virtual sensor model of my application with deep learning is the possible amount of customization and its vast predictive capabilities. Also, neural networks were used in the past to solve time

series tasks [36, 94, 150, 153]. Various extensions to the classical neural network have been made since its introduction. This chapter introduces the most relevant basic and extensive algorithms for this part: the artificial neural network, gradient descent, the convolutional neural network (CNN), the recurrent neural network (RNN), the long short-term memory network (LSTM), batch normalization, dropout, and approximate model uncertainty.

I partly base this chapter on following journal article:

> Oehmcke, S., Zielinski, O., and Kramer, O. (2017a). Input quality aware convolutional LSTM networks for virtual marine sensors. *Neurocomputing*, 275:2603–2615.

5.2 Artificial Neural Networks

A basic artificial neural network consists of an input layer, various hidden layers, and an output layer. The classical fully connected or dense layer is described as:

$$\hat{\boldsymbol{y}}_t := \phi(\boldsymbol{b} + W \cdot \boldsymbol{x}_t) \tag{5.1}$$

with a layer input vector $\boldsymbol{x}_t := (x_1, \ldots, x_d)^T \in \mathbb{R}^d$, output vector $\hat{\boldsymbol{y}}_t := (\hat{y}_1, \ldots, \hat{y}_{d'})^T \in \mathbb{R}^{d'}$, weight matrix W, bias vector $\boldsymbol{b}$, activation function ϕ, and time $t \in (1, \ldots, n)$. The number of patterns n depicts how large the dataset is. The weight matrix has a shape equal to the number of inputs d by the number of outputs/neurons d': $W \in \mathbb{R}^{d' \times d}$. Further, the length of the bias vector is equal to the number of neurons: $\boldsymbol{b} \in \mathbb{R}^{d'}$. Then, the next layer uses the current output $\hat{\boldsymbol{y}}_t$ as its new input $\boldsymbol{x}_t$ with new weights W and bias $\boldsymbol{b}$. The last layer in a network, the output layer, usually employs a linear activation function $\phi_{lin}(\boldsymbol{q}) := \boldsymbol{q}$ for an input vector $\boldsymbol{q}$ in regression tasks. The number of neurons d' in this linear layer amounts to the number of regression targets the model is going to predict. A complete model f consists of multiple *simple* layers that together can approximate

difficult functions. The model predicts an unknown output vector $\boldsymbol{y}_{t'}$ for a previously unseen input vector $\boldsymbol{x}_{t'}$: $f(\boldsymbol{x}_{t'}) := \hat{\boldsymbol{y}}_{t'} = \boldsymbol{y}_{t'} + \epsilon$, with an error ϵ. When the model is trained, this error ϵ is minimized by optimizing a loss function, e.g. the mean squared error (MSE) ($L_{MSE} : 1/n \sum_{t=1}^{n} (\boldsymbol{y}_t - \hat{\boldsymbol{y}}_t)^2$), to gain optimal weights [125]. Through changes of Equation 5.1, special layers are possible, such as convolutional or recurrent layers.

5.3 Optimizing with Gradient Descent

To find the optimal values for the weights and the biases of a neural network, gradient descent is most commonly applied. The approach calculates the pull towards an optimum of a model function f for input x [12, 51]. The function could be the loss function and the input could be the network inputs and their weights. This pull is called gradient, which is calculated from the derivatives of the function f with respect to the input x:

$$\frac{\partial f(x)}{\partial x} := \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \,, \tag{5.2}$$

with a theoretically infinitesimally step size h. The gradient shows the direction and intensity of change if the input x is altered with the step size h. If the input is a vector $\boldsymbol{x}$, the derivatives are called partial derivatives.

There are two basic approaches to calculate the gradient: Firstly, the numerical gradient that guarantees to create a better solution with an appropriately small step size h. It is inefficient to compute as every dimension of the current input vector $\boldsymbol{x}$ needs its own evaluation of function f (forward pass) to measure the effect of the chosen step size h. This approach can still be useful if the function f is a blackbox because no information about its derivatives are required. The second approach, the analytical gradient, is faster, because it calculates all gradients at once in one forward pass through the network. Here, we require full knowledge of the

function f and all operations must be differentiable. This is not a problem for neural networks as they are designed only with known operations, such as multiplications or summations.

Let us look at an example for analytical gradient descent: Assume we have an input vector $\boldsymbol{x} = (2,4)^T$, a target output $y = 0$, a bias $b = 2$ and an initial weight vector $\boldsymbol{w} = (5,-2)$. This is a simple network, similar to the Rosenblatt perceptron [122], has no hidden layers and only linear activation, but the algorithm does not change with more layers and different activation functions as long as the derivatives are known. We want to minimize function $f(\boldsymbol{x}, y, \boldsymbol{w}, b) = |y - (b + \boldsymbol{w}\boldsymbol{x})|$. All derivatives for an input q of the individual operations are known:

$$\begin{array}{llll} \frac{\partial}{\partial x} & q + z = 1 & \frac{\partial}{\partial q} & x \cdot z = z \\ \frac{\partial}{\partial x} & |q| \quad = \frac{q}{|q|} & \frac{\partial}{\partial q} & -q = -1 \end{array} \quad ,$$

with another value z. First, we perform a forward pass through this small network and save all values for the individual operations: $f(\boldsymbol{x}, y, \boldsymbol{w}, b) = 4$. Figure 5.1 shows the forward pass values in green. Then, the gradient at the networks output is set to either 1 for maximization or -1 for minimization, because we pull the weights into the direction of our optimization task. In this case, we want to minimize. Now, the partial gradients for every operation in function f are calculated from inner to outer operations and connected via the chain rule. Every partial gradient depends only on the previous gradient because of the chain rule and if the operation requires it, also on its forward pass values. This mechanism explains why it is also called backpropagation, since the error is propageted backwards through the function (backward pass). The red values in Figure 5.1 depict those gradients. The adjusted weights $\boldsymbol{w}'$ and bias b' are then:

$$\begin{array}{lll} \boldsymbol{w}' & += & \partial \boldsymbol{w} \cdot l \\ b' & += & \partial b \cdot l \ , \end{array} \tag{5.3}$$

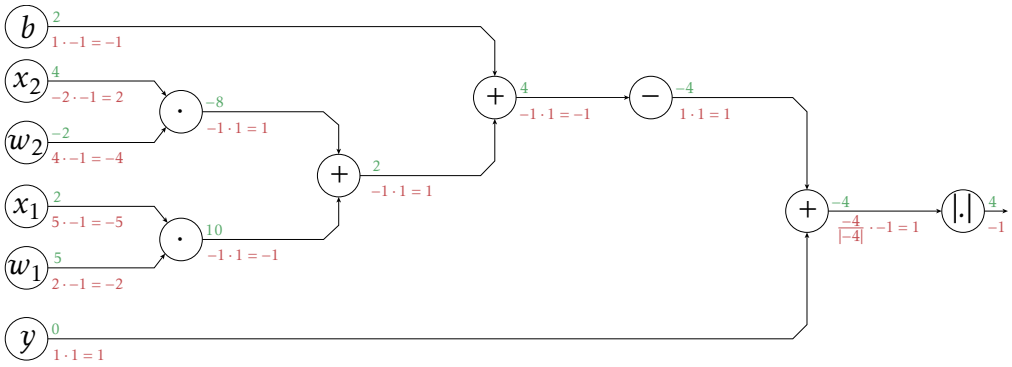


Figure 5.1: Example of a single gradient descent step. The forward pass is colored green while the backward pass is red.

with the gradient weight vector $\partial \boldsymbol{w} = (-2, -4)$, gradient for the bias $\partial b = -1$, and learning rate $l = 0.1$. After a new forward pass we are already closer to the optimum: $f(\boldsymbol{x}, y, \boldsymbol{w}', b') = 1.9$. Note that only the weights get adjusted and not the input vectors, because we try to model the target outputs with our chosen network layout. If these steps are repeated multiple times, eventually the minimum is reached if the learning rate is appropriate. When the learning rate is too small, learning can take many steps but will reach some optimum, even though it could be a local optimum. If the learning is too large, the steps towards the optimum can be too large and results can even get worse.

Until now, I only considered a single pattern, but if a dataset has more than one pattern, the question arises how to process the complete dataset of n patterns to find the optimum fast while staying accurate. The stochastic gradient descent (SGD) iterates pattern by pattern through the dataset and applies the gradient update to the weights after each pattern [85]. In contrast, the batch approach calculates the gradient for one iteration by processing the complete dataset. SGD needs less memory and it is shown that it reaches the same optimum as the true gradient after a sufficient number of iterations. Moreover, the mini-batch learning approach is also iterative, but the gradient in one iteration is calculated from m patterns,

where $1 < m < n$. Depending on the dataset, available memory, and implementation this can be faster because fast vectorization operations can be used. Also, it potentially converges faster, because the noise of one pattern does not influence the gradient as much. If the batch size is too large, the model might not generalize as well as with smaller batch sizes because it convergences to sharp minimizers of the training function [78].

An extension is the addition of a momentum term to retain some consistency in the optimization [117]:

$$\begin{aligned} \boldsymbol{v} &= \mu \cdot \boldsymbol{v} - \partial \boldsymbol{w} \cdot l \\ \boldsymbol{w}' &+= \boldsymbol{v} \end{aligned} \tag{5.4}$$

The weights cannot be updated directly but are subject to a velocity vector $\boldsymbol{v}$ which is zero at the beginning. This is helping the learning process to account for extreme gradients that might also be outliers. The new hyper-parameter μ slows down the current velocity and can be understood as friction parameter. Without friction the optimizer could not be able to stop at the optimum and overshoot.

In this work, I use the Adam optimizer [79]. Adam is similar to another optimizer, RMSprop [140], but with momentum:

$$\begin{aligned} \boldsymbol{m} &= \beta_1 \cdot \boldsymbol{m} + (1 - \beta_1) \cdot \partial \boldsymbol{w} \\ \boldsymbol{v} &= \beta_2 \cdot \boldsymbol{v} + (1 - \beta_2) \cdot \partial \boldsymbol{w}^2 \\ \boldsymbol{w}' &+= -\frac{l \cdot \boldsymbol{m}}{\sqrt{\boldsymbol{v}} + \epsilon} \end{aligned} \tag{5.5}$$

As standard, the authors give following parameter values: $\epsilon = 1\mathrm{e}{-8}, \beta_1 = 0.9, \beta_2 = 0.999$. The parameters β_1, β_2 balance the rate of decay and added velocity of the vectors $\boldsymbol{m}, \boldsymbol{v}$. At the beginning of the learning process, the weight updates are bigger and then get smaller over time while depending on the gradient. Adam is at the time of writing one of the most used

optimizers and shows competitive or superior results in many applications. It is also per-parameter adaptive, which means that it holds an individual learning rate for each weight.

5.4 Specialized Learning Layer

The dense network has been adapted for specific tasks. Here, I present the ones used in my architectures.

5.4.1 convolutional neural network (CNN)

A downside of dense layers is that the known order of data cannot be taken into account easily. Such an order can a the arrangement of pixels in an image or the temporal order of a time series. Dense layers could still learn to perform their task, but it will take some time as they do not consider the underlying pixel or temporal order. One approach that takes advantage of this order is filtering by convolution. Prior to the success of CNNs in the computer vision domain [82, 132] and later in other domains, filters were hand-crafted by application experts. For time series tasks, the filter feature $\hat{\boldsymbol{y}}_{\check{t}}^{conv}$ is extracted by applying a filter/kernel matrix K at time step $\check{t}$ to the input matrix $\boldsymbol{x}$:

$$\hat{\boldsymbol{y}}_{\check{t}}^{conv} := conv(\boldsymbol{x}_{\check{t}}) = \boldsymbol{b} + \sum_{i=1}^{|K|} \boldsymbol{x}_{\check{t}-\frac{|K|}{2}+(i-1)} \boldsymbol{k}_i \ . \tag{5.6}$$

The kernel size $|K|$ reflects how many other inputs around $\check{t}$ are needed to compose the filtered output. If several $conv$ operations follow after another, they cover more time and extracted features get more abstract and meaningful. A convolutional layer (conv layer) applies this operation to the complete input series $X_{t-\delta+1:t} := (\boldsymbol{x}_{t-\delta+1}, \ldots, \boldsymbol{x}_t)$ with time frame

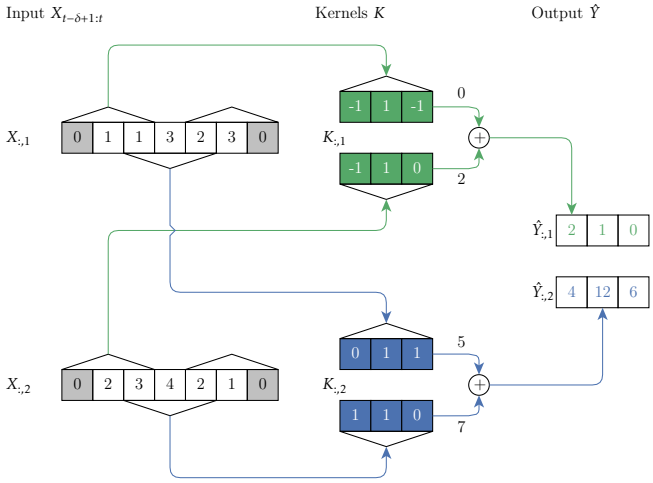


Figure 5.2: One dimensional convolutional example. Input series of length $\delta = 5$ is filtered by filters $d' = 2$ with stride $s = 2$ and kernel size $|K| = 3$.

length δ at time t and d' filters:

$$\text{conv}(X_{t-\delta+1:t}) := \phi_{conv} \begin{pmatrix} conv_1(\boldsymbol{x}_{t-\delta+1}) & conv_1(\boldsymbol{x}_{t-\delta+s}) & \dots \\ \vdots & \ddots & \\ conv_{d'}(\boldsymbol{x}_{t-\delta+1}) & & conv_{d'}(\boldsymbol{x}_t) \end{pmatrix}. \tag{5.7}$$

The stride s dictates in which frequency the operations are applied, e.g., if $s = 1$ the filter feature is computed for every time step, but with $s = 2$ the output is reduced to $\delta/2$. Most networks use the rectified linear unit ($ReLU$) function ($\phi_{ReLU}(\boldsymbol{x}) := \max(\boldsymbol{x}, 0)$) [103] as activation function ϕ_{conv}.

Figure 5.2 shows a short example calculation. Assume there is an input tensor, with $\delta = 5$ time steps and $d = 2$ channels. This input is processed by $d' = 2$ kernels with stride of $s = 2$ and kernel size of $|K| = 3$. Kernel one and its flow of values is depicted in green and for the second filter blue color is used. The green example focuses on the first time step for the output, while the blue example illustrates this for the kernel outputs second time step.

In terms of computational efficiency, the conv layer needs less trainable weights than a dense layer. Because the conv layer applies the same kernel on the whole time series ($|K| \times d$), while the dense layer requires a weight for every input feature as well as every time step ($\delta \times d$). The number of filters of the conv layer correspond to the number of output channels d',. At the beginning and end of a series, zero padding is used, which inserts zero values at the series to create a valid convolutional operation. Problems occur if the time steps are not equidistant, e.g. measured with varying sampling rates, because the extracted features can only be detected within a certain margin in a stretched or squeezed time series.

5.4.2 RNNs and LSTMs

Another way to incorporate temporal information is to process a series sequentially with a recurrent neural network layer [53]. In contrast to conv layers, this approach does not rely on data points with equidistant time steps between them. The input series X_t^δ is sequentially processed vector for vector in δ steps. At some time step $\check{t} \in (t - \delta + 1, \ldots, t)$ the output is defined as:

$$\begin{aligned} \hat{\boldsymbol{y}}_{\check{t}}^{RNN} &:= \phi_{RNN}\left(\boldsymbol{b}_y + W_{hy}\boldsymbol{h}_{\check{t}}\right) \\ \boldsymbol{h}_{\check{t}} &:= \phi_h(\boldsymbol{b}_h + W_{xh}\boldsymbol{x}_{\check{t}} + W_{hh}h_{\check{t}-1}) \ , \end{aligned} \tag{5.8}$$

whereas the hidden state $h_{\check{t}}$ acts as a memory that introduces previous inputs back into the network with a hidden state weight W_{hh} and bias $\boldsymbol{b}_h$. The usual weight and bias from Equation 5.1 is her denoted as W_{hy} and $\boldsymbol{b}_y$. The hidden states activation ϕ_h is usually a logistic curve ($sigmoid$) function and the outer activation often is the hyperbolic tangent ($tanh$) function.

Although this works well for short time series, longer time series are prone to create the vanishing gradient problem because the same weight

matrix is multiplied at each time step. This is a problem for the training process because when the gradient is smaller than it should be, adjusting the weights towards the optima is as slow as choosing a too small learning rate would be. Further, events that happened some time steps ago are only weakly reflected in the hidden state. LSTM layer [60] were introduced to counter these problems. First applied to language tasks [52, 54, 135], but also more general time series tasks [37, 38, 46] fit naturally with the LSTM layer. A gating mechanism helps to retain temporal distant information while keeping the gradient at a steady size. The LSTM algorithm updates the hidden state $\boldsymbol{h}$ at every time step with the element-wise-product ($\circ$) of an output gate $\boldsymbol{o}$ and the activation of a cell $\boldsymbol{c}$. This output gate learns how to write the input. Inside the cell is the forget gate $\boldsymbol{f}$ that dictates how much information of the cell is kept from its previous iteration. The second part of the cell is the product of the input gate $\boldsymbol{i}$ and the input modulation $\boldsymbol{j}$, which learns how to read the input. All gates consider the hidden state from the preceding time step.

In full detail, the LSTM is defined as follows:

$$\begin{aligned}
\hat{\boldsymbol{y}}_{\check{t}}^{LSTM} &:= \phi_{LSTM}\left(\boldsymbol{b}_y + W_{hy}\boldsymbol{h}_{\check{t}}\right) \\
\boldsymbol{h}_{\check{t}} &:= \boldsymbol{o}_{\check{t}} \circ \phi_h(\boldsymbol{c}_{\check{t}}) \\
\boldsymbol{c}_{\check{t}} &:= \boldsymbol{f}_{\check{t}} \circ \boldsymbol{c}_{\check{t}-1} + \boldsymbol{i}_{\check{t}} \circ \boldsymbol{j}_{\check{t}} \\
\boldsymbol{i}_{\check{t}} &:= \phi_i\left(\boldsymbol{b}_i + W_{xi}\boldsymbol{x}_{\check{t}} + W_{hi}\boldsymbol{h}_{\check{t}-1}\right) \\
\boldsymbol{f}_{\check{t}} &:= \phi_f\left(\boldsymbol{b}_f + W_{xf}\boldsymbol{x}_{\check{t}} + W_{hf}\boldsymbol{h}_{\check{t}-1}\right) \\
\boldsymbol{o}_{\check{t}} &:= \phi_o\left(\boldsymbol{b}_o + W_{xo}\boldsymbol{x}_{\check{t}} + W_{ho}\boldsymbol{h}_{\check{t}-1}\right) \\
\boldsymbol{j}_{\check{t}} &:= \phi_j\left(\boldsymbol{b}_j + W_{xj}\boldsymbol{x}_{\check{t}} + W_{hj}\boldsymbol{h}_{\check{t}-1}\right) .
\end{aligned} \tag{5.9}$$

With help of the subscripts, the affiliations of activations ϕ, biases $\boldsymbol{b}$, and weight matrices W can be traced. The activation function for the gates are again usually the $sigmoid$ function and the output activation is the $tanh$ function. I present a visual comparison of dense, RNN, and LSTM layers

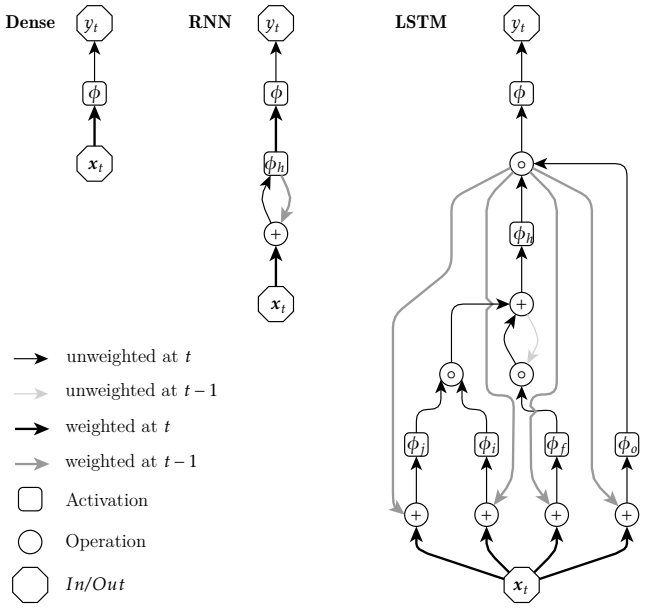


Figure 5.3: Graphical comparison of dense, recurrent neural network, and long short-term memory network layers.

in Figure 5.3 that highlights the increased complexity of LSTM layer, but also emphasizes on how the gates are interacting.

When two independent LSTM layers read the input sequence from two different directions, we speak of a bidirectional long short-term memory network (bLSTM) [52, 129]. This extension concatenates the output of two LSTMs, where one LSTM reads the input from front to back (forward LSTM) and one reverses the input sequence (backward LSTM). Often the backwards LSTM is inferior performance-wise, but the combined output of the forward and backward LSTM can outperform an ordinary LSTM. A limitation of this approach is that is not applicable to online learning since the complete sequence is not available.

5.4.3 Combining CNN and LSTM layer

The combination of conv and LSTM layers seem like a natural fit. First, meaningful features get extracted from parts of time series or images by

conv layer. Then, the recurrent or LSTM layer process these features to infer the information of the entire pattern. This has already been used in many applications. For example, Girshick *et al.* [48] for object detection and segmentation; Sainath *et al.* [126] on various large vocabulary speed recognition tasks; Tsioroni *et al.* [142] for gesture recognition in images; Zhou *et al.* [157] in a community question answering task; and Morales *et al.* [101] for recognition of activity with mobile wearable sensors. Other recurrent layer methods, such as gated recurrent units [25], have also been employed after conv layers, for instance in spatial audio tagging [149]. The bLSTM layer also was used in such fashion in a tool wear and tear scenario by Zhao *et al.* [155].

Another way to combine conv and recurrent layers is to mix the layers. Liang *et al.* [88, 89] realize this by including recurrent connections in the conv layer. They employ this new layer in a static object recognition task. Similar to this but with a gated RNN, Shi *et al.* [130] include the convolutional operation directly into the LSTM layer. This allows a better representation of spatio-temporal correlations in video datasets. They achieved good results on a moving MNIST and Radar Echo dataset. Pinheiro and Collobert [115] introduce the recurrent connections between conv layers in an image scene labeling task. This approach was also employed on multivariate time series data with prior clustering of the inputs [150].

5.5 Mechanisms to Support Learning

Several mechanisms and heuristics have been proposed in the literature to optimize the learning process. This section describes the most important ones applied in this work.

5.5.1 Modules

In practice, there are layer configurations that have proven to be effective for specific tasks [3]. These configurations reduce the difficulty of finding an optimal architecture for a network. They can be seen as tried and tested building blocks. A well-known module is the inception module by Szegedy *et al.* [136]. With these modules they build the GoogLeNet architecture that won the ImageNet challenge 2014. One inception module consists of four parallel lanes: the first lane is a single 1×1 conv layer, the second is a 1×1 followed by a 3×3 conv layer, the third is the same, but the last conv layer has a kernel size of 5×5, the last lane is a 3×3 max pooling followed by a 1×1 conv layer. All outputs of these lanes are concatenated. In contrast to the relatively high number of weight parameters in an inception module, the SqueezeNet architecture by Iandola *et al.* [68] utilizes fire-modules. These aim to balance a low number of weight parameters and good prediction performance. To that end, only conv layers with a kernel length of 1×1 or 3×3 are utilized. First comes one 1×1 conv layer to "squeeze" the input, which then gives its output to another 1×1 and a parallel running 3×3 conv layer. Lastly, these two layers outputs are concatenated. A schematic representation of a fire module for one dimensional data is given in Figure 5.4. As a side note, a 1×1 conv layer is used to reduce the input dimension and also adds more non-linearity, while a bigger kernel length creates non-linear time dependent features.

5.5.2 Batch Normalization

Deep neural network architectures use many layers and the parameters of these are trained with gradient descent [51, Chapter 8.7.1]. Although these parameter updates should be independent for each layer, they are not, because the gradient flows through the whole network. This dependency introduces negative higher-order effects on early layers with a smaller order near the output. In shallow networks, this is not a problem as the effects

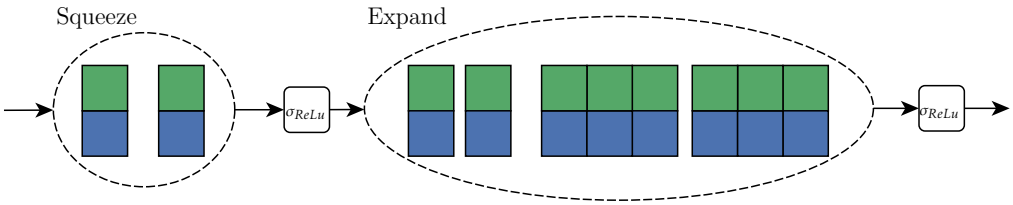


Figure 5.4: Schematic representation of an example fire-module. It contains a squeeze layer with two filters. The expand layer has two filters with kernel size of one and another two with kernel size three. The number of channels d is two.

are negligible, but the deeper the network the stronger the effects. For instance, if a gradient is relatively small, the previous layer also gets a smaller gradient and so it gradually vanishes. Moreover, an initially large gradient can cause an explosion of the gradient values. This could result in no learning or random learning and finding a single learning rate for all layers would not be possible. To counteract these higher-order effects, one can optimize the parameters while regarding the effects (e.g. calculate the Hessian matrix [40, 47]), but this is computationally demanding for second-order effects and infeasible for higher-order effects.

Batch normalization is another solution to this problem. It is proposed by Ioffe and Szegedy [69]. Here, the outputs of layers $\hat{\boldsymbol{y}}$ get normalized per mini-batch. Usually, the batch size is much smaller than the entire number of training data n, because of restricted availability of memory and better generalization (see Section 5.3). Batch normalization is either applied before or after the outermost activation function. The gradient gets scaled by the mean $\mu_{\hat{Y}}$ and variance $\sigma_{\hat{Y}}$ of the current batch $\hat{Y} := (\hat{\boldsymbol{y}}_1, \ldots, \hat{\boldsymbol{y}}_m)$

with batch size m:

$$\begin{aligned}
\mu_{\hat{Y}} &:= \frac{1}{m}\sum_{i=1}^{m}\hat{Y}_{i,:} \\
\sigma^2_{\hat{Y}} &= \frac{1}{m}\sum_{i=0}^{m}(\hat{Y}_{i,:}-\mu_{\hat{Y}})^2 \\
\hat{Y}'_{i,:} &:= \frac{\hat{Y}_{i,:}-\mu_{\hat{Y}}}{\sqrt{\epsilon+\sigma^2_{\hat{Y}}}}
\end{aligned} \tag{5.10}$$

The ϵ is introduced to avoid the division by zero state and should be small (e.g. $10e{-}8$). This effectively approximates the Hessian matrix and thus helps with the second-order-effects.

5.5.3 Dropout

Dropout is a mechanism that avoids overfitting and also acts as a regularizer [133]. Recent publications about state-of-the-art network architectures most often use this mechanism in one or more layers [68]. The main idea is to temporarily deactivate a part of the network for one iteration. We interpret every weight matrix W as a random matrix containing the inner weight matrix $\hat{W}$ and a dropout vector $\boldsymbol{z} := (z_0,\ldots,z_{d'})^T \in \{0,1\}$ that depends on a probability value p:

$$W := \boldsymbol{z}_j \circ \hat{W} \ , \tag{5.11}$$

with $\circ$ being element-wise multiplication. Every titeration initializes a new dropout vector, otherwise the same weights would stay deactivated during training. The dropout vector acts as a switch in the gradient descent algorithm, when the value is zero the gradient is set to zero as well:

$$\frac{\partial}{\partial W}(\boldsymbol{z}\circ W)\cdot\boldsymbol{x}\cdot\partial U := \boldsymbol{z}\cdot\boldsymbol{x}\cdot\partial U \ . \tag{5.12}$$

with ∂U being the previous gradient link from the chain rule.

When dropout is used in LSTM layers, the standard definition only allows to deactivate neurons along the input weights, but not in the recurrent units. This is problematic as it leads to overfitting of the weights from the gates. Further, at every time step of an input series other neurons get deactivated which potentially reduces the gradient towards zero. This happens because the same weight matrix gets dropped at different indices over time and eventually the weight matrix's gradients are all zero. Variational dropout [43] solves this by re-parameterizing Equation 5.9:

$$\phi_{LSTM}\begin{pmatrix} \boldsymbol{i}_{\check{t}} \\ \boldsymbol{f}_{\check{t}} \\ \boldsymbol{o}_{\check{t}} \\ \boldsymbol{j}_{\check{t}} \end{pmatrix} := \phi_{LSTM}\left(\begin{pmatrix} \phi_i \\ \phi_f \\ \phi_o \\ \phi_j \end{pmatrix}\left(\boldsymbol{b} + \begin{pmatrix} \boldsymbol{x}_{\check{t}}^T \\ \boldsymbol{h}_{\check{t}-1}^T \end{pmatrix} \cdot W_{LSTM}\right)\right). \tag{5.13}$$

This representation ties the weights to a single weight matrix W_{LSTM} with dimensionality $2d$ by $4d$ with d being the dimensionality of $\boldsymbol{x}_{\check{t}}$. Now, the recurrent unit weights can be dropped and the deactivated parts remain the same during one iteration.

5.6 Predictive Uncertainty

Machine learning methods learn from the distribution they are given for training, which is often unknown before training. If a trained model is asked to predict an output with an input vector that lies *outside the training distribution*, the results may vary [42]. For example, let the target output be the temperature in an office and the input vector consists of surrounding sensors. When the model is trained only with data from during the day, nighttime predictions will lie outside of the normal distribution as the heater usually does not run and outside temperatures drop as well. The output might still be usable, depending on the model extrapolation

ability and the problem complexity. Nevertheless, the model could signal that the input lies outside the training distribution and that the output thus might be more unreliable. Moreover, if the training and test data is afflicted by noise, the *data uncertainty* increases. In our office temperature example this would appear when the sensors have a low measuring precision.

Another kind of uncertainty is *model uncertainty* and occurs in two situations. The first one appears, when there is uncertainty in the parameters or the weights, biases, and hyper-parameters in a neural network. There are many possible parameter settings that result in local optima, this raises the question which parameter settings would most suited to find a good local optima; e.g. if to rely more on one or the other sensor input for the prediction. Second, the structure of the model might be uncertain as in there might be multiple possible layer arrangements or activation function choices that are viable.

In combination, data and model uncertainty give us a sense of how confident our model is in its prediction, the *predictive uncertainty*. Uncertainty intervals are used to visualize this uncertainty. A wider interval corresponds to a higher uncertainty. These intervals can be produced by Bayesian neural networks, which are highly specialized. Unfortunately, they do not scale well and are difficult to train. The adaption to complex architectures is difficult, because of the special operations they need. An alternative is the approximation of uncertainty by using dropout and Monte Carlo (MC) passes of the network at test time. This has its theoretical bases in interpreting the dropping of values at random as an approximate Bayesian inference in a deep Gaussian process. The final output $\hat{\boldsymbol{y}}$ of the output layer corresponds to the predictive mean $\mathbb{E}[\boldsymbol{y}]$ of n^{MC} MC model realizations $f_i \in F$:

$$\mathbb{E}[\boldsymbol{y}_t] \approx \frac{1}{n^{MC}} \sum_{i=1}^{n^{MC}} f_i(\boldsymbol{x}_t) \; . \tag{5.14}$$

Additionally, we can give a predictive variance $\widetilde{\mathrm{Var}}[\boldsymbol{y}_t]$:

$$\widetilde{\mathrm{Var}}[\boldsymbol{y}_t] \approx \frac{1}{n^{MC}} \sum_{i=1}^{n^{MC}} g_i(\boldsymbol{x}_t) + f_i(\boldsymbol{x}_t)^2 - \mathbb{E}[\boldsymbol{y}_t]^2 \ . \tag{5.15}$$

These approximations get more accurate and stable with an increasing number of MC models n^{MC}. It is important that all MC models f_i realize a different dropout vector $\boldsymbol{z}$ per iteration, otherwise there would not be any variance. Process or observation noise g is either a static value or a dynamic function (heteroscedastic uncertainty). By taking the square of the predictive variance we get the standard deviation of the predictions, which equals a 68.27% uncertainty interval. A 95.45% uncertainty interval would be: $2 \cdot \sqrt{\widetilde{\mathrm{Var}}[\boldsymbol{y}_t]}$.

5.7 Conclusion

Deep learning has become a wide field for research. In this chapter, I covered the most important algorithms and approaches for this work. These include the basic dense layer, the specialized conv layer, and recurrent layer. Further, I gave insight into the optimization with gradient descent and its variations. I also described approaches to help the training of a network with modules, batch normalization, and dropout. Finally, I introduced the predictive uncertainty through dropout. With this toolset, I am ready to build deep learning architectures for the given marine time series application.

6

Virtual Sensors with exPAA and bLSTMs

6.1 Introduction

The foundations of deep learning from the previous chapter can now be applied to build the virtual sensor for the flow sensor from the surrounding sensors. Virtual sensors consist of physical or data-driven models. I employ the data-driven models because physical or physically-based numerical models often take a long time to build or compute and require extensive knowledge about underlying processes. The advantage of data-driven models such as machine learning models is that they rely on objective information, i.e. the observed data. These models must be able to handle the challenges of this data as well, such as noise and outliers.

In this chapter, I introduce a deep architecture based on bLSTM layers combined with a time dimensionality method to replace the defective BEFmate flow sensor from Section 2.3.1 with a virtual one. Because LSTM

layers are specifically designed for sequential data and inherit a high predictive power, I choose a network architecture with these layers. The time dimensionality reduction is necessary since computations on longer time series get increasingly expensive since more recursions are performed. I extend piecewise approximate aggregation (PAA) [76] by assuming that the most recent time steps are more important for the model prediction and should be less compressed. The extended method is named exponential piecewise approximate aggregation (exPAA). In experimental analyses, I compare PAA and exPAA representations on the combined BEFmate and TSS dataset. Further, I test if the resulting models remain stable in their prediction over time.

The structure of this chapter is as follows. Related work is discussed in Section 6.2. The time dimensionality reduction PAA and exPAA are presented in Section 6.3. Section 6.4 introduces the recurrent network architecture for the virtual sensor. An experimental analysis is presented in Section 6.5, while conclusions are drawn in Section 6.6.

This chapter is partly based on the following published paper:

> Oehmcke, S., Zielinski, O., and Kramer, O. (2017b). Recurrent neural networks and exponential PAA for virtual marine sensors. In *International Joint Conference on Neural Networks (IJCNN)*, pages 4459–4466. IEEE.

6.2 Related Work

The usage of artificial neural networks and other machine learning algorithms in the sensor and similar domains is no novelty [50, 75, 94, 101, 150, 152, 153]. For instance, a study on the performance of neural networks and nearest neighbor algorithm in an underwater application has been conducted by Baladrón *et al.* [7]. They have been comparing both machine learning methods in an imputation and prediction task regarding

their mean errors. Different combinations of sensors have been removed for the imputation task, while the prediction task entailed a 24-hour forecast. They concluded that neural networks are a good choice for their marine application, but the performance of the nearest neighbors method has been equally good in some cases.

The inference of gene regulatory networks with random forest has been introduced by Maduranga *et al.* [92]. They compared it to Bayesian networks as well as ordinary differential equations and showed superior results. By dividing the multi gene prediction problem into smaller problems, they created easier learning problems that could be solved separately. Since the application usually does not feature noise or bias in their data, which is a crucial difference to my marine data.

A robust framework for general inference problems in sensor networks has been proposed by Paskin *et al.* [112]. It is based on junction trees, which is a data structure that supports the modeling of sensors as nodes. To achieve global consistency, it inherits local consistency between adjacent nodes. This approach proves to be robust against temporary failure of nodes and is capable to mitigate effects of unreliable communication. The downside is the requirement of extensive domain knowledge that is not available for my application.

SMiLer is a framework to forecast multiple sensors by Zhou *et al.* [156]. Their forecast models are based on Gaussian processes. To select a training set, this framework employs nearest neighbor queries with DTW as distance measure, which has been implemented to run on the GPU. I cannot apply this approach as it is forecast task and not a virtual sensor task, but it is an interesting example of multivariate prediction.

Moreover, virtual sensors based on machine learning have not been employed in the marine domain until now. Usually, these virtual sensor approaches are mainly applied in fault detection scenarios [4, 38, 123, 134] and for emission predictions [70, 86, 104].

6.3 Time Dimensionality Reduction with exPAA

The LSTM layer is a powerful tool to predict time-dependent target outputs, since it utilizes a working memory. This memory comes at a cost.

With each time step, the runtime of the LSTM layer increases linearly since the gating and weighting operations are applied at each time step. This becomes a problem with a larger number of observed time steps and dimensions. Moreover, the LSTM can get distracted if too much redundant information are present, which can hinder the learning process.

A solution to the problem of long sequences is the reduction of time steps while keeping the most important information. To achieve this, time dimensionality reduction methods are applied. They reduce the number of time steps, from δ steps to δ' steps, with $\delta' < \delta$. The underlying assumption for these methods is that there are periods in a time series that are more interesting for the target prediction than others. These methods then focus on the important periods, while retaining less information about other periods. Although an LSTM could learn through its forget gate when the important parts are happening, the processing is much faster when it needs to consider fewer steps.

Most of these time dimensionality reduction methods originate from time series indexing tasks, such as PAA [76], adaptive piecewise constant approximation (APCA) [22], or discrete Haar wavelet transformation (DWT). Time series indexing is in most cases applied to univariate series ($d = 1$). If applied to multivariate series, the time reduction would not be the same for all dimensions, e.g. one dimension is reduced at the beginning of the series and another dimension is reduced at the end. For example, APCA creates more parts at periods that have a lot of value changes. This creates time series with shifts in between the features and is computationally ineffective as all dimensions would need separate calculations. LSTM layers can handle time series that have time steps which are not equidistant in time, but these unrestricted inner shifts would make learning harder.

PAA does not depend on the time series itself, but only on δ and δ'. It creates the same representation for all dimensions in a time series, disregarding the actual values. Although the computational costs are low, the

results are still competitive to more complex methods such as DWT or singular value decomposition. The input series $X_{t-\delta+1:t} := (\boldsymbol{x}_{t-\delta+1}, \ldots, \boldsymbol{x}_t)$ is reduced to $\bar{X}_{t-\delta+1:t} := (part_1, \ldots, part_{\delta'})$. This reduced series consists of parts $part_j$ with $j \in (1, \ldots, \delta')$:

$$part_j := \mathsf{agg}\left([\boldsymbol{x}_{t-k}]_{k=split_j}^{split_{j+1}} \right) , \tag{6.1}$$

with an aggregation function agg that often is the mean function. To acquire the splits, PAA calculates:

$$split_i := \delta \cdot \frac{i}{\delta'} , \tag{6.2}$$

with $i \in \{1, \ldots, \delta' + 1\}$. Interestingly, the representation of PAA is identical to that of a non-overlapping pooling layer. While the kernel size of pooling [51, Chapter 9.3] is fixed to ensure that parts always contain the same amount of values, PAA can have differently sized parts depending on the splits.

The split calculation of PAA assumes that a reduction with equal sized parts is best to retain information for the target output prediction. This is not true for many prediction tasks as the most recent values usually have the highest impact. To align with this new assumption, I introduce exPAA. It changes the split calculation to:

$$split_i := round\left(\delta \cdot \frac{i}{\delta'}^{e} \right) , \tag{6.3}$$

with an exponent e. This essentially creates parts with different amount of values. The normal PAA representation can still be built by choosing the exponent e as 1. When the exponent is greater than one, the more recent values get represented with more parts. Consequently, if the exponent is less than one, past values are finer approximated through more parts. An edge case occurs if two following splits contain the same indices due

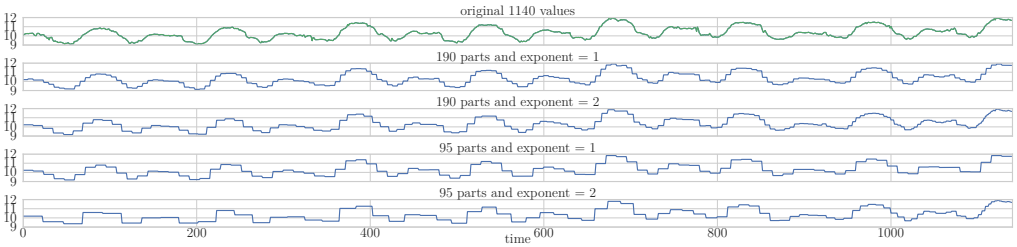


Figure 6.1: Applied exPAA on temperature time series. There are finer approximations at the end of the series when exPAA utilizes an exponent greater than one.

to rounding, which would result in an empty part. To avoid this, the subsequent split is increased by one. In the worst case, this could create multiple one-value parts, but this is preferable to empty or undefined parts and is possible as long as $\delta > \delta'$ is true. The influence of the exponent can also be seen in Figure 6.1.

The idea of finer approximating recent values also resonates with the exponentially weighted rolling average method [66]. Here, the entire signal is smoothed with fading weights for past values. In contrast to this method, I want to reduce the time steps to process less information and not necessarily smooth the time series.

6.4 Network Architecture

I employed the following architecture: First, the preprocessed data is passed to an exPAA layer that uses the average as aggregate function agg for the individual parts. Then, these parts are given to three bLSTM layers that are connected after another, with 280 neurons in the first layer and respectively 480 neurons for the following two. The inner activation of these layers utilize the *tanh* function, while the recurrent connections are activated through the *sigmoid* function. The first and second bLSTM layer return their sequence, but the last layer only returns its last step. This output is processed by a dense layer with 480 neurons with *ReLU* ac-

tivation. Before each hidden layer, dropout is performed with 0.1 dropout probability for inputs, 0.5 for bLSTM layer, and 0.1 for the dense layer. The output layer employs L2 regularization of 0.0001. I employ Adam to optimize the learning of weights. The batch size is 256.

The hyper-parameter search is done according to [14, 125]. First, I optimize with only five epochs on the training set. Then, I repeat this process with the five best models and 40 epochs. The validation error is measured every second epoch on a previously unseen set of the training data. Validation sets are sampled uniformly and additionally contain the next 14 hours after a random point to guarantee a full tidal cycle per validation pattern.

6.5 Experimental Analysis

In the following, I analyze the influence of exPAA on the combined dataset of TSS and BEFmate (see Section 2.4).

6.5.1 Design

My two hypotheses for these experiments are: The bLSTM architecture benefits from exPAA by decreasing runtime and testing error. My model of the defective sensor delivers stable output approximations over time.

The training set consists of values from 2014-09-18 15:00 until 2015-03-31 22:40:00. This corresponds to 60% of the data and includes 6979 steps of the target sensors as well as 24922 steps of the surrounding sensors. These numbers are so different because the target sensor measures only at a high water level, but the surrounding sensors measure continuously. I append up to 24 hour of data to each target input step. Optimization of the bLSTM hyper-parameters is done by further dividing the training set into 70% for training and 30% for validation. After optimal hyper-parameters were found, I trained the network on the complete training set.

The remaining 40% or 4654 target sensor steps of the dataset are the test set. Because of the different scaling of the target sensors, I create a virtual sensor for each of the five target sensors instead of a single one.

The free parameter values for this experiment are the original window size $\delta \in \{24, 42, 84, 144\}$, reduced window size $\delta' \in \{4, 10, 42\}$, and the exponent $e \in \{1, 1.5, 2\}$. I choose these original window size settings, because they approximately correspond with the tidal and daily cycles: 4, 7, 14, and 24 hours. The baseline is an architecture without exPAA or windowing. This changes the bLSTM layers to fully connected layers as only one time step is considered. The scenario where $\delta = \delta'$ is not tested, except for $\delta = \delta' = 42$ because preliminary tests did show inferior performance and the runtime is infeasible. I set the number of runs to 30 per condition.

To measure performance, I employ the root mean squared error (RMSE):

$$\text{RMSE} := \sqrt{\text{mean}(Y - f(X))^2} \tag{6.4}$$

and the root median squared error (RMedSE):

$$\text{RMedSE} := \sqrt{\text{median}(Y - f(X))^2}\ . \tag{6.5}$$

While the RMSE takes the performance peaks or lows into account, the RMedSE is robust to these extreme values and favors generalizable models that occasionally produce outliers.

6.5.2 Results

The results for the best free parameter settings ($\delta = \{42, 84\}$, $\delta' = \{4, 10\}$) are presented in Table 6.1. This shows that the dimensionality reduction of time steps is beneficial since the models always deliver sub-optimal performance without temporal reduction ($\delta = \delta' = 42$). For the three sensors *Direction*, *Pressure*, and *Speed* the results with exPAA are always better than the baseline. Moreover, only the RMSE for *Speed* is lowest with the

Table 6.1: Comparison of exPAA settings with regard to RMSE and RMedSE. The lowest error of a sensor is bold.

		Direction				*Pressure*				*Speed*			
orig. size δ =		42		84		42		84		42		84	
window size δ' =		4	10	4	10	4	10	4	10	4	10	4	10
metric	e												
RMSE	–	71.297	71.297	71.297	71.297	2.526	2.526	2.526	2.526	6.673	6.673	6.673	6.673
	1	70.677	74.863	76.536	75.016	2.555	2.551	2.595	2.589	6.446	6.793	**6.132**	6.389
	1.5	70.096	75.462	73.887	76.732	2.540	2.498	**2.369**	2.403	6.335	6.481	6.319	6.489
	2	71.151	76.315	**69.007**	75.393	2.552	2.478	2.371	2.405	6.308	6.208	6.276	6.447
RMedSE	–	24.599	24.599	24.599	24.599	1.813	1.813	1.813	1.813	4.455	4.455	4.455	4.455
	1	24.868	22.191	25.893	24.519	1.734	1.699	1.780	1.731	4.101	4.379	3.897	4.142
	1.5	24.324	22.146	25.776	23.788	1.682	1.627	1.615	1.692	3.964	3.913	3.947	4.001
	2	24.473	**22.104**	23.459	22.933	1.687	1.615	**1.546**	1.635	3.961	**3.896**	3.927	4.071

		Conductivity				*Temp*			
orig. size δ =		42		84		42		84	
window size δ' =		4	10	4	10	4	10	4	10
metric	e								
RMSE	–	**2.133**	**2.133**	**2.133**	**2.133**	**2.422**	**2.422**	**2.422**	**2.422**
	1	2.362	2.583	2.336	2.746	2.635	2.725	2.760	2.827
	1.5	2.340	2.614	2.326	2.714	2.676	2.766	2.714	2.781
	2	2.338	2.575	2.298	2.703	2.654	2.767	2.730	2.843
RMedSE	–	**1.544**	**1.544**	**1.544**	**1.544**	**1.509**	**1.509**	**1.509**	**1.509**
	1	1.785	2.050	1.856	2.351	1.894	2.007	1.979	2.290
	1.5	1.814	2.161	1.830	2.285	1.888	2.064	2.033	2.237
	2	1.803	2.184	1.807	2.256	1.947	2.108	1.988	2.274

exponent being 1. In the other cases higher exponents are better, which supports the first hypothesis that exPAA helps to achieve lower error rates. Interestingly, *Temp* and *Conductivity* sensor models were better without any windowing or PAA method.

The feasible combinations of original and reduced window size are either $\delta = 84$ and $\delta' = 4$ or $\delta = 42$ and $\delta' = 10$. This is a trade-off between more past information in a more compact form or finer approximated steps that reach less far into the past. In contrast to RMSE models, the models chosen with RMedSE prefer the smaller window size, which could mean that these smaller window sizes are good for generalization, but suffer from occasional outlier predictions. This is supported by the observation that the RMedSE is always lower than the RMSE. I expected this, since the median is more robust against outliers.

In Figure 6.2, I compare the best settings for exPAA with the non PAA approach. All measured differences can be significant, if the notches do not overlap [23]. I confirmed this observation with a statistical test, the one-sided Mann–Whitney U test [95, 98], since the p-value is below 0.05 and the observed U value is lower or equal than the critical U value ($U \leq 450$). The *Conductivity* plot is especially interesting, because the whiskers of the exPAA approach show a strikingly lower error. For the *Temp* target sensor there exists at least one outlier with equivalent performance compared to the non PAA approach. This means that my approach can be better, but with the current setups the learning does not always converge to its optimum.

Compared to the standard deviation of the target sensors on the test set, the results are also good. For example, the standard deviation of the *Direction* sensor is 100.743, while the RMSE is 69.007 and the RMedSE is 22.104. As the water is flowing mainly from either 54 or 244 degree (see Figure 2.5), there is a clear distinction between the two with the current error rates. The other target sensors are more difficult to interpret, but the er-

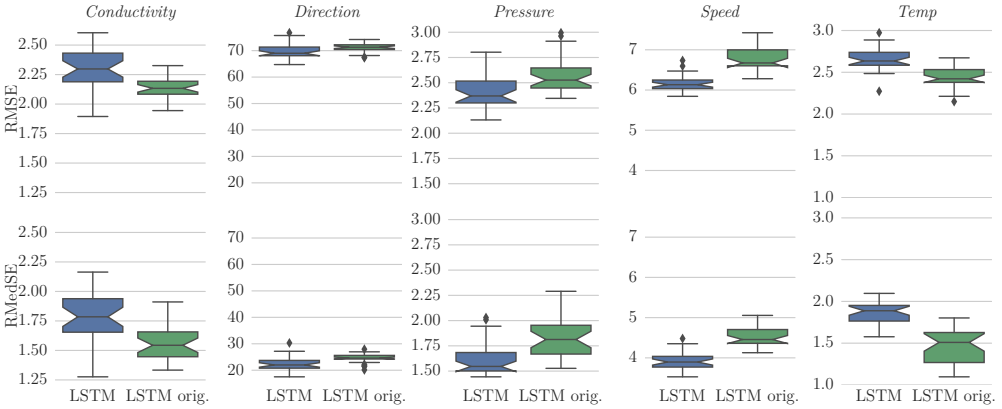


Figure 6.2: Box plots of best exPAA settings performance for the bLSTM architecture in comparison to their baseline prediction without exPAA windows (LSTM orig.).

ror is always lower than the standard deviation: *Speed* (std: 6.582, RMSE: 6.132, RMedSE: 3.896), *Pressure* (std: 2.698, RMSE: 2.365, RMedSE: 1.546), *Temp* (std: 4.266, RMSE: 2.422, RMedSE: 1.509), and *Conductivity* (std: 4.015, RMSE: 2.133, RMedSE: 1.544).

To analyze the stability over time of my predictions, I divide the test set into five equal sized parts. Further, to be able to compare the parts, I divide the RMSE and RMedSE by the corresponding standard deviation of the parts. The performance per part is presented in Figure 6.3. Although there are differences between the parts, only the error of the *Temp* model seems to increase over time. The *Conductivity* prediction again shows a huge variance in its performance, especially in the first three parts. In general, most models stay stable over time in their prediction error.

6.6 Conclusion

Building a deep learning architecture for a new application is challenging. The possible design space is large, ranging from number and type of layer to employed activation function or number of neurons. In this chapter, I presented the first iteration of a bidirectional LSTM architecture for

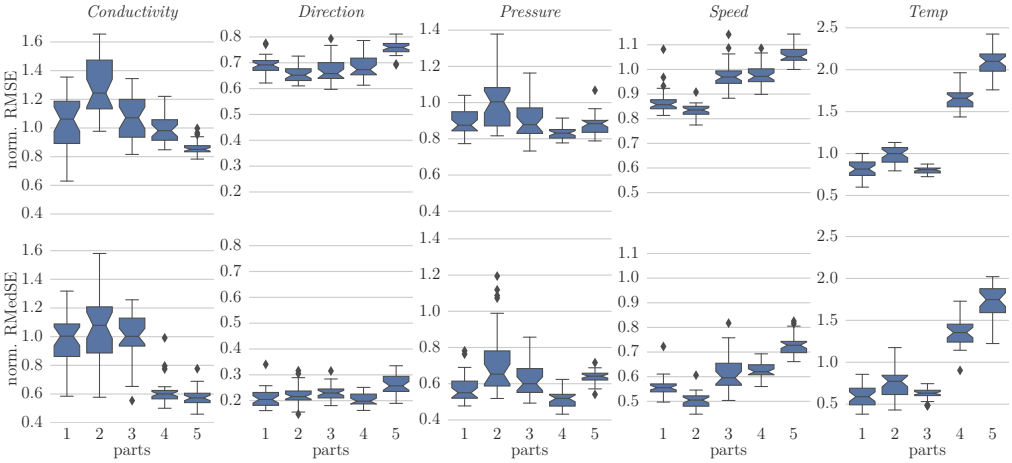


Figure 6.3: Box plots about the five parts of the validation set. Depicted is the prediction performance of the bLSTM architecture in comparison to their baseline prediction without PAA.

the virtual sensor modeling of the BEFmates defective flow sensor. The model inputs are the surrounding sensor measurements of the combined TSS and BEFmate dataset. I proposed exPAA, a novel time dimensionality reduction method based on PAA, which represents more recent values in greater detail, but past values are still available.

The results of my experiments demonstrate that exPAA reduces computation time and also increases the performance of my network in almost all cases. These experiments compared the PAA and exPAA representations to an architecture without time representation. Moreover, models with exPAA are stable throughout the five-month period of the validation set. These results show great potential for an initial architecture, but there is room for improvement.

7

Improved Architecture with CNNs, Uncertainty Predictions, and Input Quality

7.1 Introduction

The previous chapter presented a virtual sensor based on a stacked bLSTM architecture to benefit from the time series structure of my marine application. Together with the novel time dimensionality reduction method exPAA, low error rates are achieved for this model, which will now be my baseline model. Inspired by the recent success of CNN architectures, I know that meaningful non-linear features can be extracted from patterns with automatically learned filters. These filters can potentially assist the LSTM to learn problem relevant information faster. Further, up to this point, only point predictions were used, but uncertainty intervals could indicate how reliable the model is at a given time step. Eventually, the

baseline model uses the imputed dataset, but does not account for the introduced bias by the imputed values even though this information is available.

In this chapter, I present an improvement to the network architecture of the baseline model with a focus on three main contributions: First, I introduce the automated filtering of inputs with convolutional modules (fire-modules [68]) before the bLSTM layer. The second contribution is the predictive uncertainty as proposed by Gal *et al.* [42]. In the third contribution, I employ the input quality information from the penalized DTW*k*NN imputation in a novel input quality based dropout (qDrop) layer . This qDrop layer improves the robustness of a model while decreasing the mean prediction error. I verify the new architecture called *MarineNet* in an experiment that encompasses a comparison to the baseline model and an analysis of the qDrop layer w.r.t. the point and uncertainty interval prediction errors.

The chapter is structured as follows. At first, I present the new dropout layer considering the input quality in Section 7.2. Then, the complete architecture of the machine learning-based marine sensor model is introduced in Section 7.3. I present an experimental analysis in Section 7.4. Finally, I address some of the shortcomings of my architecture through an revision in Section 7.5 and conclude this chapter in Section 7.6.

This chapter is partly based on following journal article:

> Oehmcke, S., Zielinski, O., and Kramer, O. (2017a). Input quality aware convolutional LSTM networks for virtual marine sensors. *Neurocomputing*, 275:2603–2615.

7.2 Accounting for Known Input Quality

Although I use the DTW*k*NN ensemble method to impute the missing value as introduced in Chapter 4, a bias is introduced by every imputation

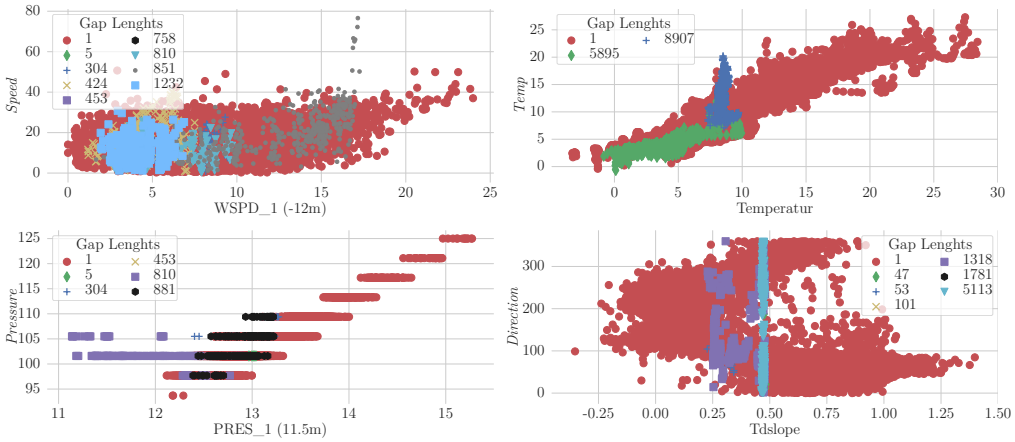


Figure 7.1: Influence of poorly imputed values. On the x-axes are input sensor features, while the target sensors are on the y-axes. Different marker symbols represent different gap lengths.

method. The longer the gaps, the higher the bias introduced to the data. I plot the target sensor values against the partly imputed sensor values in Figure 7.1. There, the marker symbols represent the different gap lengths. Longer imputed gaps in the data distort the original correlations. If a learning algorithm uses these imputed time series, it also unknowingly learns its bias. Classic algorithms such as kNN are able to apply weight changes that incorporate information about the input quality because they compare distances. For neural networks this is more complicated, even though I hold the input quality information in form of the penalty matrix P from the imputation method.

I propose a new dropout layer, called qDrop. It applies an individual feature-wise dropout with dropout vector $\boldsymbol{p}_t^{quality} := 1 - P_{t,:}^{\varepsilon}$ with dropout probability values at each time step t instead of a global dropout probability value p. The original penalty matrix P consists of very small feature values for larger gaps, because it contains the inverse of the gap length. This is unfavorable, because the input would get dropped very often. For example, even a gap length of 2 would have a dropout probability of 0.5,

which corresponds to dropping the value half the time. To counter this effect, the exponent ε regulates the values of P, with $0 \leq \varepsilon \leq 1$. The dropout mechanism is well suited for this, because it can avoid an overfitting to untrustworthy values and see changes in the predictive uncertainty.

When the network is in training phase, it will drop the imputed values based on the probability vector $\boldsymbol{p}_t^{quality}$, which in return reduces the influence of these biased values. Normally, dropout is not applied at test time because the network would not be able to utilize all its neurons. This creates results inferior to those of the complete network. In contrast, the predictive uncertainty, as introduced in Section 5.6, employs MC predictions with enabled dropout that are equal or better than the prediction of the full network. With MC predictions, I can drop inputs without risking completely losing the feature information. Because the qDrop layer drops the input features based on the input quality, those imputed values are useful. But they also have a smaller impact on the prediction as they get ignored depending on their dropout probability. As the dropout rate increases with input of lower quality, the standard deviation of MC prediction also increases if the feature is important for the prediction target. This standard deviation ties directly to the predictive uncertainty.

Interestingly, it is often advised that after the input layer the probability value for dropout should be zero or only small [42, 133]. This case is different because the dropped features are not selected completely at random and instead of only one or three features, 56 are available. The qDrop layer does not drop all essential features at once unless they are all part of a long gap, because of the existing redundancies.

7.3 MarineNet: Architecture

An overview of the network architecture, which I call MarineNet, is given in Figure 7.11. At first, the input layer receives the imputed time series.

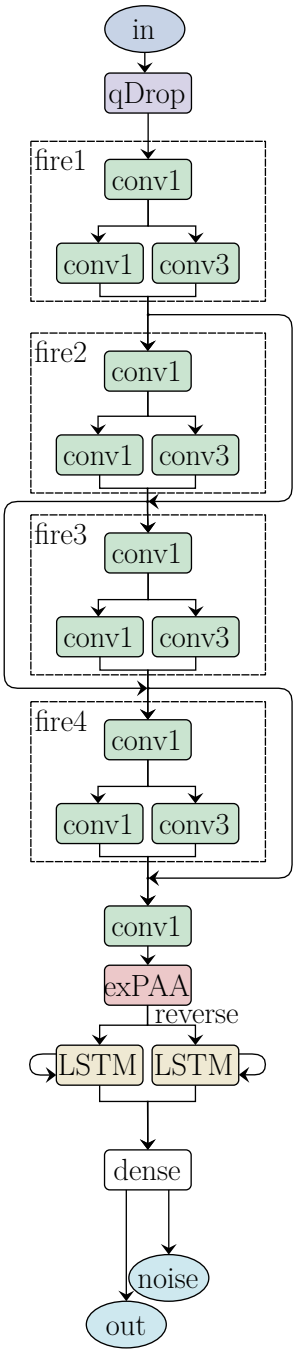


Figure 7.2: The macroarchitectural view of MarineNet. The number behind the conv layers depicts the width of a kernel.

Table 7.1: On the left side is the MarineNet architecture with number of parameters (Param.#) and depth (D.). On the right side is the baseline model from the previous chapter. The variables δ and δ' are the original and reduced time frame length. Non-trainable parameters are from batch normalization, since the weights are not adjusted by gradient descent.

MarineNet				Baseline bLSTM			
Layer	Output	D.	Param#	Layer	Output	D.	Param.#
input	$\delta \times 57$	0	0	input	$\delta \times 57$	0	0
qDrop	$\delta \times 57$	0	0				
fire1	$\delta \times 128$	2	15 904	exPAA	$\delta' \times 57$	0	0
fire2	$\delta \times 128$	2	19 312				
fire3	$\delta \times 128$	2	19 312	bLSTM1	$\delta' \times 240$	δ'	69 920
fire4	$\delta \times 128$	2	19 312				
conv1	$\delta \times 128$	1	17 024	bLSTM2	$\delta' \times 480$	δ'	923 520
exPAA	$\delta' \times 128$	0	0				
bLSTM	512	δ'	788 480	bLSTM3	$\delta' \times 480$	δ'	1 384 320
dense	512	1	264 704	dense	480	1	230 880
out g	1	1	512				
out f	1	1	512	out f	1	1	480
Total params			1 145 072				2 709 120
Trainable params			1 142 384				2 709 120
Non-trainable params			2688				0
Total depth			$12 + \delta'$				$2 + 3 \cdot \delta'$

Then follows the qDrop layer, which applies dropout to values biased by the imputation method (see Section 7.2). Next, four fire-modules are sequentially employed. Similar to the architecture in Chapter 6, an exPAA layer processes these feature time series to have fewer time steps, but detailed information are still present for the most recent time steps. A bLSTM layer handles the reduced series. Only the last recurrent output is forwarded to a dense layer. Finally, there are two output layers. One represents the target sensor regression. The other one is modeling the process noise. I

present a detailed comparison of MarineNet to the architecture of Chapter 6 in Table 7.1. It shows that MarineNet requires 1 145 072 parameters instead of 2 709 120, which is a factor of 2.365 less. Also, as I use a single bLSTM layer instead of three, the total depth of the network depends less on the amount of reduced time steps δ'.

A new addition to this architecture is, besides my qDrop layer, the usage of conv layers. They can be seen as preprocessing for the bLSTM, because they obtain useful features from the time series by learning feasible combination of sensors and value movements over time. To have good predictive performance in an affordable time span, I decided to use fire-modules (see Section 5.5.1). Other modules, such as the inception module have a significantly higher number of parameters to train.

I designed the convolutional part of the network in such a way that the created features can contain a time frame of up to 90 minutes (9 time steps). One single fire-module summarizes a 30-minute time span into its features, because of the kernel width of 3 in one of the conv layers and the datasets 10-minute resolution. Every following fire-module incorporates 20 minutes (2 time steps) more into the last convolutional output, because the features always overlap to ⅔ with another. Figure 7.3 visualizes this. Note that the series length remains the same after each convolution, as zero padding is applied and the input patterns include more than nine time steps. So four fire-modules result in $30 + 3 \cdot 20 = 90$ minutes ($3 + 3 \cdot 2 = 9$ time steps).

To help the network converge more quickly, the SqueezeNet authors [68] were inspired by He *et al.* [58] to use shortcuts. A shortcut passes the output of one fire-module not only to the next module, but also to the one after that. They found an increase in performance without introducing new trainable weights. I use shortcuts between fire-modules, which allows a skipping of layers that are difficult to train until the previous layers are trained.

The available data for training and testing is limited to one year. This

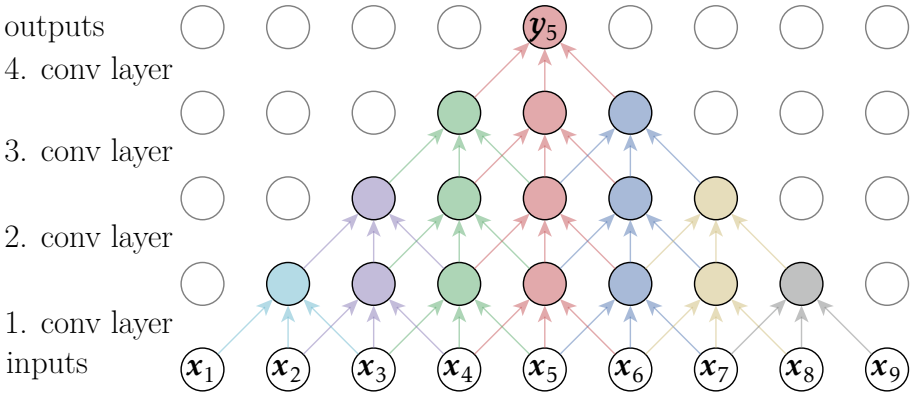


Figure 7.3: Simplified example of the output of four conv layers with a kernel width of 3. It shows that the last layers output for the fifth step contains information of nine time steps.

poses a problem for the reliability of my model, because it can encounter pattern configurations that are only sparsely represented in the training data. By employing predictive uncertainty (see Section 5.6), I know when a model is uncertain about its prediction. I assume heteroscedastic uncertainty since the noise in this application is not static. For example, changing tides and seasons create varying noise signals. The noise equals the inversion of the model precision. This dynamic observation noise is denoted as function $g(\boldsymbol{x}_t)$ and is included into the loss function of the training phase:

$$L := \alpha \cdot (\boldsymbol{y}_t - f(\boldsymbol{x}_t))^2 \cdot (g(\boldsymbol{x}_t) + 1) - (1 - \alpha) \cdot \log(g(\boldsymbol{x}_t)) \ , \tag{7.1}$$

with a trade-off variable $\alpha \in [0, 1]$ for easier uncertainty calibration. This is necessary, because the target output and noise can have different scales and characteristics, which the original constant 0.5 can not account for. The first part of the equation is the quadratic loss multiplied by the noise, while the second part encourages the noise to grow. In contrast to the original equation from Gal *et al.* [42], I made some practical adjustments. Although important for theoretical implications, in practice the constant

part ($^{d}/_{2} \log 2\pi$) can be left out, since the dimensionality d is static. Since rounding to zero errors appeared, I added a plus one to the noise at the quadratic loss part to prevent a numerical overflow of the noise function. Note that the noise function must always be greater than 0, because negative noise makes no sense, which is why I applied the $softplus$ activation [35] to the noise output layer. More calibration is applied after the training with linear calibration.

Finally, here is a brief overview of the smaller details of the architecture: I apply $ReLU$ activation for conv and dense layers. For the bLSTM layer, I use $tanh$ as output and $sigmoid$ as inner activation functions. After an activation, there is always a batch normalization layer. Also, the outputs of the two separate LSTM layers are concatenated. Dropout is employed before every layer with trainable weights and set to a dropout probability of $p = .25$, except after the input layer. Also, all trainable layers use a L2 regularization of $9e{-}11$. The exPAA layer uses max as agg function, which returns the maximal value of every feature across one part. I replaced mean aggregation with max, because the features from the prior conv layers are most prominent with high values. For pooling other researcher also found that max gives superior performance [49, 102]. Since exPAA breaks the time series equidistant of time steps, I used a bLSTM layer instead of more convolutions. Also, the bLSTM layer offered a good prediction performance in the previous architecture. The network uses no pooling layers before conv layers, because 1D convolutions are fast to compute and I do not want to lose information early in the network. To adjust the gradient, I choose Adam [79] as optimizing algorithm with a learning rate of $l = 0.01$. The chosen batch size is 256. Every prediction is calculated by $n^{MC} = 150$ MC runs.

7.4 Experimental Analysis

I analyze the performance of MarineNet on the same combined dataset of TSS and BEFmate as in Section 6.5. In this section, I describe the design of the experiment, how I optimized hyper-parameters, and compare MarineNet to the baseline bLSTM model from the previous chapter. The focus lies on the impact analysis of the qDrop layer.

7.4.1 Design

The data is divided into the same 60%/40% parts as described in Section 6.5.1. This is important for a fair comparison between my new architecture with the previous bLSTM architecture. My qDrop layer is tested with different settings for the quality exponent: $\varepsilon \in \{.0, .03125, .0625, .125, .25, .5, 1, \mathsf{inf}\}$. I mainly consider exponents smaller than one, since the values of the quality input matrix P are small, and exponents higher than one would increase the dropout chance even further. Moreover, the setting $\varepsilon = .0$ performs no dropout at the qDrop layer and will be the standard model to compare to. If $\varepsilon = \mathsf{inf}$, the imputed values are dropped entirely.

Each model is trained for 200 epochs, whereby the final weights are chosen w.r.t. the lowest error on the validation set. A model chooses its validation set uniformly at random from the training set. In addition to the random validation time steps, the next 144 steps (24h) are also selected. This time frame represents two tidal cycles and helps to predict all phases of the cycle, not only the most often. To be able to run statistical tests with sufficient reliability, I repeated each run 40 times.

For the analyses, I apply two measurements. The point prediction performance is measured with the RMSE (see Equation 6.4). I also take into

account the standard uncertainty interval:

$$\frac{1}{n}\sum_{t=1}^{n}\sqrt{\widetilde{\mathrm{Var}}[y_t]}\,, \tag{7.2}$$

with number of patterns n and the predictive variance $\widetilde{\mathrm{Var}}[y_t]$ from Equation 5.15. The performance of the uncertainty interval in combination with the point prediction can be represented with the Brier score [21]:

$$\text{Brier score} = \frac{1}{|\boldsymbol{i}|}\sum_{i\in\boldsymbol{i}}(\mathrm{acc}(i)-i)^2\ , \tag{7.3}$$

with the mean accuracy $\mathrm{acc}(i)$, which defines the percentage of values that should fall within the Gauß distribution of errors. I calculate the mean accuracy at percentage $i \in (0,1)$ over all n patterns:

$$\mathrm{acc}(i) = \frac{1}{n}\sum_{t=1}^{n}\left(|y_t - \mathbb{E}[y_t]| \leq \sqrt{\widetilde{\mathrm{Var}}[y_t]}\cdot\sqrt{2}\cdot\mathrm{erf}^{-1}(i)\cdot s\right)\,, \tag{7.4}$$

with erf^{-1} being the inverse Gauss error function and s being a scaling factor that is set to one in the beginning. The examined percentages are: $\boldsymbol{i} = (.55,.6,.65,.7,.75,.8,.85,.9,.95,.999)$. I am not interested in intervals with less than 50% accuracy, since they are equal or worse than random. Moreover, 100% accuracy is also uninteresting since a large interval will always cover all targets. The Brier score is especially important to analyze the uncertainty calibrations. A smaller Brier score, standard uncertainty interval, or RMSE reflect a better performance for their respective property.

In addition to finding a good loss/uncertainty trade-off, I calibrate the

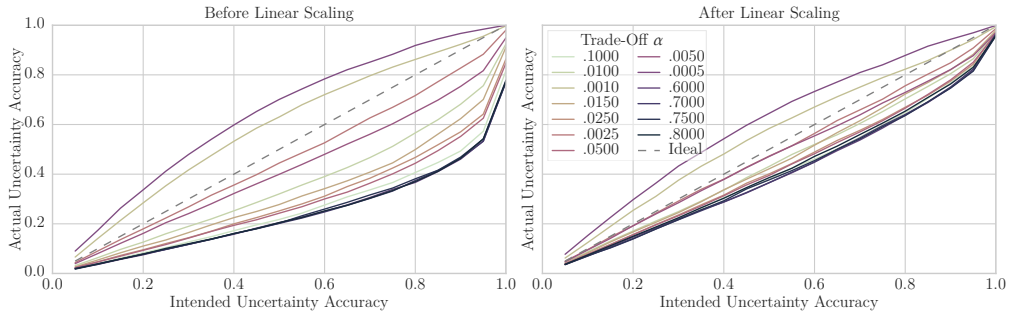


Figure 7.4: On the left side, different settings for the loss trade-off variable α for the speed sensor in a calibration plot are shown. On the right side, the linearly scaled uncertainty is presented. Note that # is short for "number of".

model on the training data by setting the scale variable s to:

$$s = \frac{1}{n_{train}} \sum_{i \in \boldsymbol{i}} \frac{\text{acc}(i)}{i} \ . \tag{7.5}$$

Both, the Brier score and the uncertainty interval are tested with and without scaling, which are referred to as *scaled* and *unscaled*.

Again, the one-sided Mann–Whitney U test is employed as statistical test. When the results contain differences with a p-value below 0.05 and a respected critical value, I call it *significant*. The critical U value with 40 runs for each condition, a standard score z of 1.64, and a significance level of 0.05, is 629.07.

7.4.2 Hyper-parameter Optimization and Calibration

I adhere to the best practices for acquiring good hyper-parameter settings from [14, 125]. To that end, I only optimize on the training data, whereby the first 70% are used for training and the rest for validation. The models for optimization are trained in only 40 epochs and with 30 repetitions.

The first step is the choosing of exPAA parameter settings. I analyze each target sensor individually. Following parameter values are tested and

Table 7.2: Choice of optimized parameter settings for MarineNet.

Sensor	#Steps δ	#Parts δ'	Exponent e	Trade-Off α
Speed	144	18	2.0	.0025000
Temp	72	18	2.0	.8000000
Conductivity	72	8	1.5	.7500000
Pressure	144	18	2.0	.9999975
Direction	144	8	1.5	.0000050

then chosen w.r.t. the lowest RMSE: time frame width $\delta \in \{18(3h), 42(7h), 72(12h), 82(13.6h), 144(24h)\}$, reduced steps $\delta' \in \{4, 8, 10, 18, 42\}$, and exponent $e \in \{1, 1.5, 2\}$.

The second optimization calibrates the prediction uncertainty to correspond to the expected uncertainty with the loss trade-off parameter α. Here, I select the parameter that produces the model with the lowest Brier score. Figure 7.4 presents the impact on the scaling for various values of α. My optimized parameter choices are shown in Table 7.2.

7.4.3 Comparison to The Baseline Model

Figure 7.5 displays a comparative box plot of the RMSE from the MarineNet and the bLSTM architecture. The increase in performance is 5.62% for *Speed*, 52.16% for *Temp*, 10.39% for *Conductivity*, 23.24% for *Pressure*, and −1.73% for *Direction*. These values are all significantly lower, except for *Direction*, where no statistical difference can be observed. Nevertheless, the minimal RMSE for *Direction* is more than five degrees lower with MarineNet. This indicates a higher potential for good performance, which can not be achieved consistently with the current configuration.

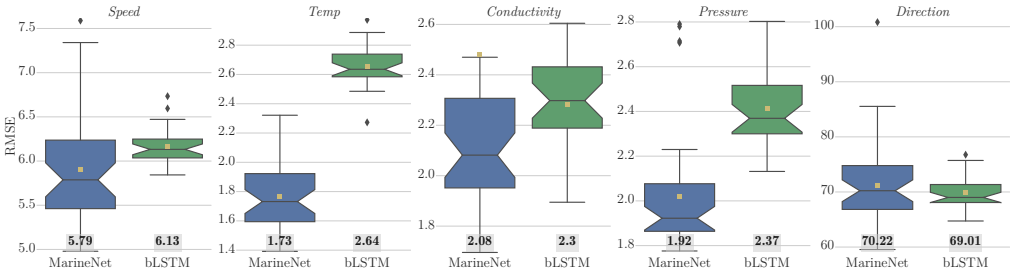


Figure 7.5: Comparing the RMSE of MarineNet and the baseline models. The median error values are written in gray boxes.

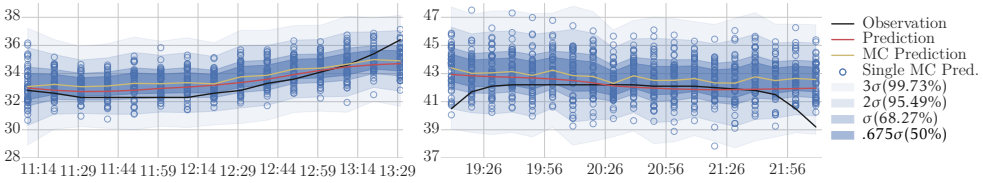


Figure 7.6: Visualized uncertainty for *Conductivity* measurements. The left side has high and the right side lower confidence.

7.4.4 Analysis of Predictive Uncertainty

To highlight the predictive uncertainty, I show two different *Conductivity* predictions in Figure 7.6. These plots are comparable, although they have different value ranges because both predictions are in a value range of eleven. If compared to the training distribution of *Conductivity* (see Figure 2.5), I realized that the first plot is closer to the training distribution than the second. This explains the larger distribution of individual MC runs and the resulting greater uncertainty interval band. A further observation on the predictive uncertainty is the small model noise at the beginning and end of a measurement cycle. This is depicted in Figure 7.7. I assume that this is correlated with the fact that most tidal cycles have similar initial and final conditions. For example, the water level is rising from zero to a maximum flood height and then falling back to zero.

In my analysis the performance of linear scaling is neither good nor bad (see Figure 7.8). While the standard uncertainty interval is smaller,

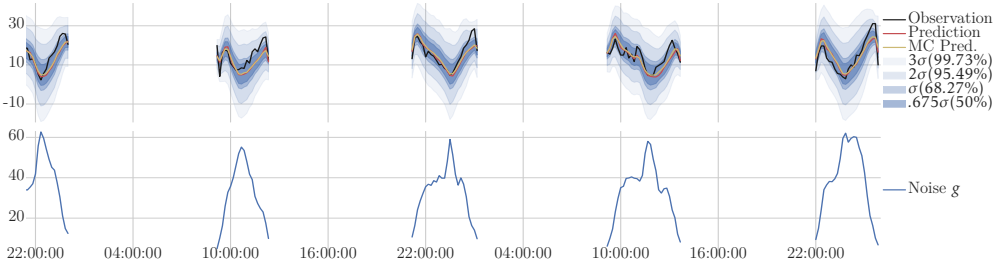


Figure 7.7: The top presents an uncertainty prediction example for the *Speed* measurements. The bottom shows the corresponding noise function g.

the Brier score is larger for the *Speed* models. The results for *Temp* and *Conductivity* are consistently negative. Again, this could be caused by the different densities in training and test sets (see Figure 2.5), which then lead to overfitting. Consistently positive, however, are the linear scaling results for *Pressure*. For *Direction*, only the standard uncertainty interval is larger.

7.4.5 qDrop Layer Analysis

In Figure 7.8 the different settings for the quality exponent ε are shown. In general, the models perform worst that completely drop the imputed values ($\varepsilon = inf$). These models perform equally to other models only in a few cases. The comparison of the models with qDrop ($\varepsilon \in \{.03125, .0625, .125, .25, .5, 1\}$) with the standard models that drops no inputs is more interesting ($\varepsilon = .0$). RMSE performance is significantly better for three sensors in six cases: *Speed* $\times 1$, *Temp* $\times 4$, *Pressure* $\times 1$. The RMSE for *Conductivity* and *Direction* never shows significant differences. The scaled standard uncertainty interval provides values that are significantly lower in nine cases for three sensors: *Speed* $\times 5$, *Pressure* $\times 1$, *Direction* $\times 3$. For the *unscaled* width of the standard uncertainty interval, nine occurrences show significantly lower values for three sensors: *Speed* $\times 6$, *Pressure* $\times 1$, *Direction* $\times 2$. In seven cases, the scaled Brier score is significantly lower for four sensors:

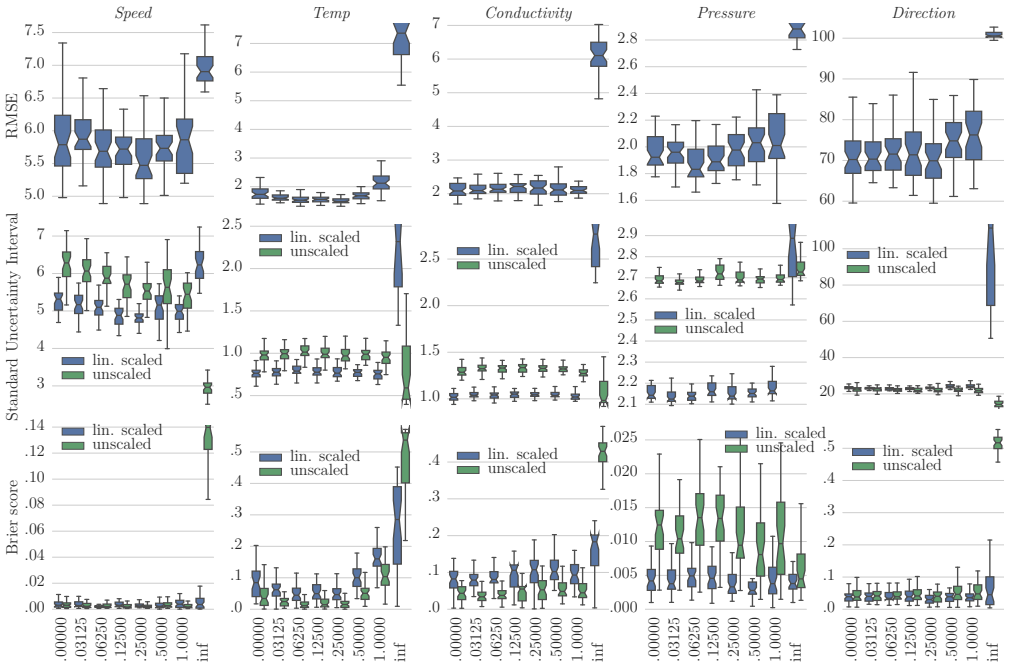


Figure 7.8: Comparing performance of qDrop settings in a box plot. The columns show the target sensors and the rows different quality measurements.

Speed ×1, *Temp* ×4, *Pressure* ×1, *Direction* ×1. Seven times the *unscaled* Brier score is significantly better for three sensors: *Speed* ×2, *Temp* ×4, *Pressure* ×1. It is important to note that the models without input quality based dropout could in no case perform significantly better than the models with qDrop. Their results are either worse or equal to qDrop models. Especially interesting is that the standard model is always beaten for the *unscaled Speed* sensor model with $\varepsilon = .25$.

For the stability analysis, I divided the test set into five parts. The results are presented in Figure 7.9. I compared the qDrop model with the lowest RMSE to the model trained without qDrop. Only the unscaled standard uncertainty interval and Brier score are utilized because the scaled results in Figure 7.8 showed dissatisfying results. To have a more meaningful RMSE for the parts, I normalized it by dividing with the standard deviation of the part values. For *Speed* and *Pressure*, the Brier score and

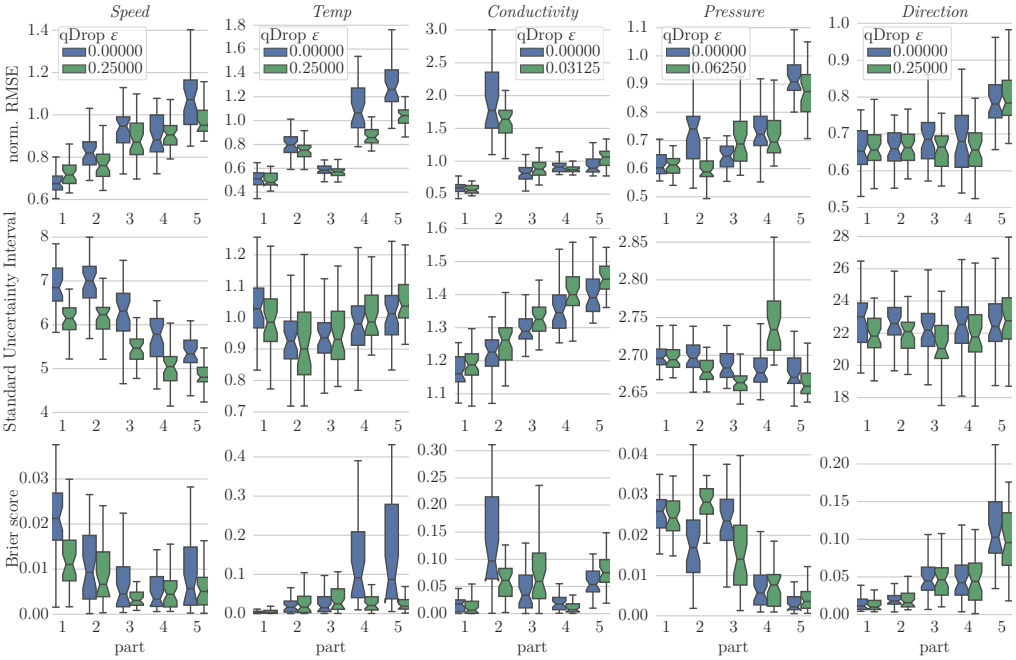


Figure 7.9: Box plots about the five parts of the validation set. Depicted is the prediction performance of the MarineNet architecture in comparison to the baseline architecture. The columns show the target sensors and the rows different quality measurements.

standard uncertainty interval are decreasing, while the RMSE increases. This is interesting because the accuracy increases, although I have a narrower prediction band and a poorer point prediction. The *Temp* models are relatively stable in Brier score and standard uncertainty interval, but the normalized RMSE is fluctuating. *Conductivity* is similar, but the RMSE fluctuations are so strong, the Brier score also becomes irregular. I theorize that these irregularities are due to the sensor drifts that are more frequent in these parts. The standard uncertainty interval for the *Direction* models is stable and the RMSE only deviates in the fifth part. The *Direction* measurements for the fifth part also shows a significantly higher standard deviation σ: 95.74, 96.83, 98.45, 99.05, **109.88**.

To gain deeper insights into the trained weights, the first conv layer is well suited. This first ”squeeze” layer reduces the dimensionality in fea-

ture space and the learned filter weights offer some intuition about feature importance. To that end, I show heat maps of the learned weights in Figure 7.10. The filter weight for each sensor come from models with activated qDrop layer. I show the minimum and maximum weights over all filters as an aggregation. Moreover, the Spearman correlation of the original feature to the target sensor is displayed. Although the correlation information is not given directly to the models, they often learned them. This can be seen in the target *Pressure* sensor that has high weights and correlation with the *PRES_1 (11.5m)* sensor from the TSS, which is only missing 4.5% of its values for longer than one time step. But in another case, *Temp* does not utilize the *Temperatur* sensor heavily and this input sensor needed 26.8% of its values to be imputed. Further, there are indications for apparent non-linear correlations, because some weights are high although there is no linear correlation between them. For example the target *Speed* and the input *Tdslope* or *Height* sensors show these non-linear correlations. Dead neurons are weights that are close to zero, even after training. These dead neurons do not seem to exist in my models. Only the *Pressure* model could be afflicted, but the heat maps color scaling is dominated by the extreme high weights for *PRES_1 (11.5m)*, which complicates an analysis there.

7.4.6 Discussion

With the presented results, I can empirically support my claim that MarineNet can handle this virtual sensor task better than the baseline bLSTM model. My new architecture needs fewer trainable weights and has a lower recursion depth, but still is able to achieve lower RMSE values. Except for the *Direction* sensor, where the results are equal for the point predictions. Although the baseline model and MarineNet both use exPAA, the results suggest that my new architecture learns more effectively, with more past steps δ, and with more reduced time steps δ' than the old one. Through the analysis of the first "squeeze" layers, I found that MarineNet learns

linear as well as non-linear correlations. Another important result is the access to predictive uncertainty in order to assess the reliability of the models, but the search for the trade-off parameter α is computationally expensive.

The advantages of employing the qDrop layer over ignoring the input quality information are substantial. This reinforces my claim that qDrop can increase the models performance, because the point predictions show lower error as well as the predictive uncertainty predictions are narrower or more accurate. I could also show that input sensors with high correlations to the target sensor are less important when they are affected by a large amount of missing value gaps. In general, MarineNet prefers complete and highly correlated data. One drawback is the dependence on information about the input quality. For other scenarios, information such as decay of sensors or failure probabilities could be used.

I created models that remain stable over time, although the amount of data is limited. Even when the RMSE got worse, the Brier score and uncertainty interval are able to reflect these uncertainties. Nevertheless, more data could lead to a better model and more insights. For example, the generalization capabilities would improve with more seasonal data. These seasonal data would be particularly beneficial for *Temp* and *Conductivity* models, as they are highly dependent on the seasons.

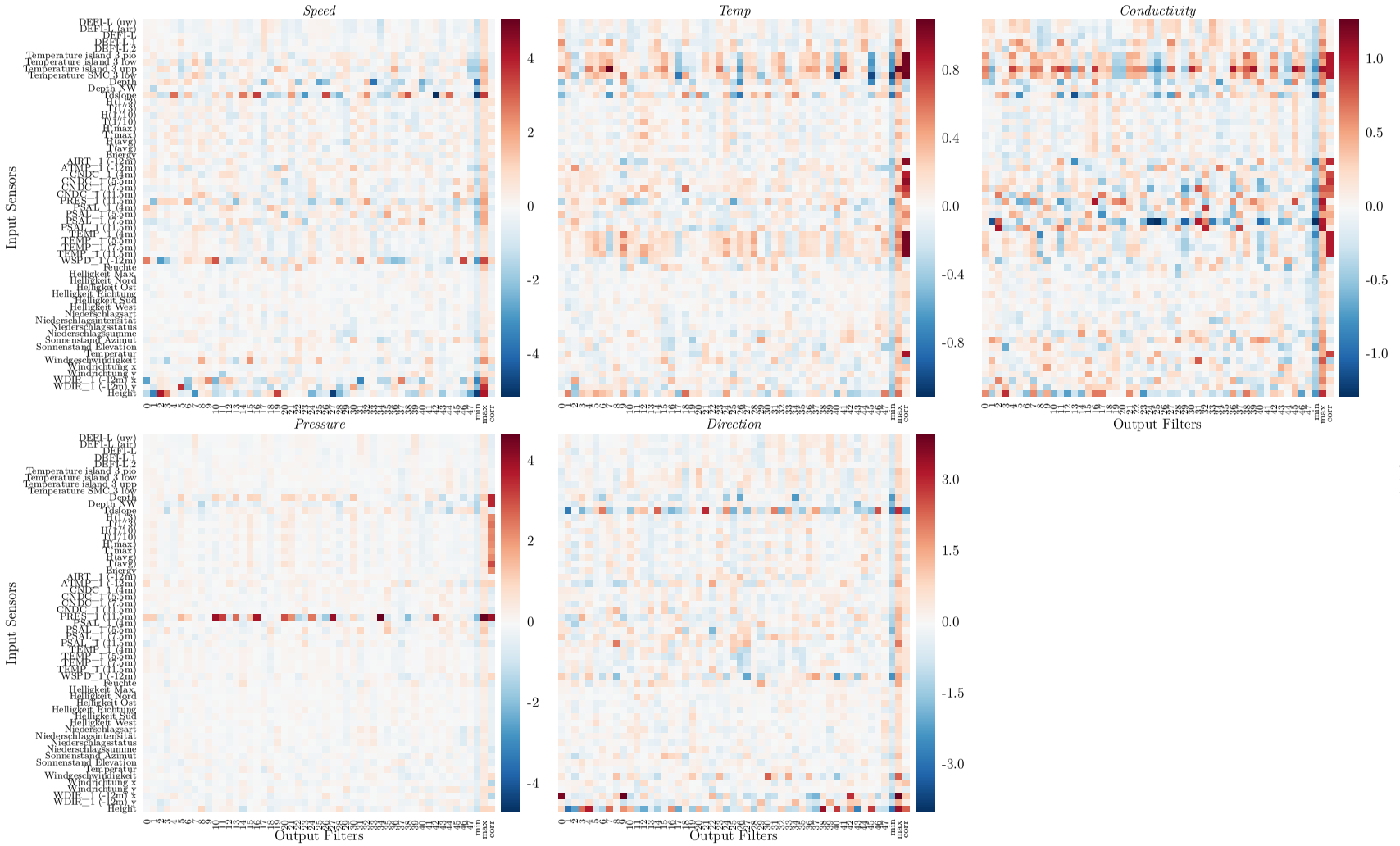


Figure 7.10: Heat map of the weights from the first conv layer of MarineNet with qDrop for each of the five target sensors. The maximum and minimum weight as well as the Spearman correlation of the target sensors with input sensors are in the far right of the heat maps. I scale the correlation value by the maximum weight value to create a coherent visualization.

7.5 MarineNet Upgrade

While analyzing the performance of MarineNet, I discovered two shortcomings. The first one is the lack of interpretability which part of the time series contributes most to the final results. This is important information as it could be the basis for the split calculations of exPAA. The second shortcoming is the required optimization for the trade-of parameter α of the loss function. To solve these, I slightly update the architecture and training process of MarineNet. Below, I present these changes and show the results in an experimental evaluation.

7.5.1 Architectural Changes

To achieve higher interpretability, I replace the dense layer by a conv layer with a kernel of size one (conv1 layer) and an averaging of the outputs over the steps. This is motivated by Iandola *et al.* [68] and Lin *et al.* [90]. A consequential change is that instead of the last output of the bLSTM layer, the complete series is propagated. Previously, this was not possible because the number of weights would have been tied to the number of exPAA parts, which I wanted to avoid. With the conv1 layer, such a restriction is not present. Interesting are the activations of this conv1 layer, as they show which part influences the prediction most. Since these activation steps are averaged to one step and then only linearly combined, one can get indications of the contributions of a step.

Since the bLSTM layer now outputs a series, I can add shortcuts after this layer. To be compatible, exPAA is applied to the shortcut outputs and then a conv1 layer balances the number of neurons. Also, additionally to the shortcut route after the fire-modules, there is another route right after the squeeze layer of each module. Further, instead of only adding the last layers output, I add all the previous shortcuts of a route to the next start of a shortcut. I drew inspiration for these dense shortcuts from Zhang *et al.* [154] and DenseNet by Huang *et al.* [63].

The number of kernels inside a fire module is now 64 for each conv layer, before the squeeze layer employed 48 neurons. I reduced the number of neurons in the bLSTM from 512 to 392. The number of neurons of the replaced dense, now conv layer remains 512. Overall, the amount of

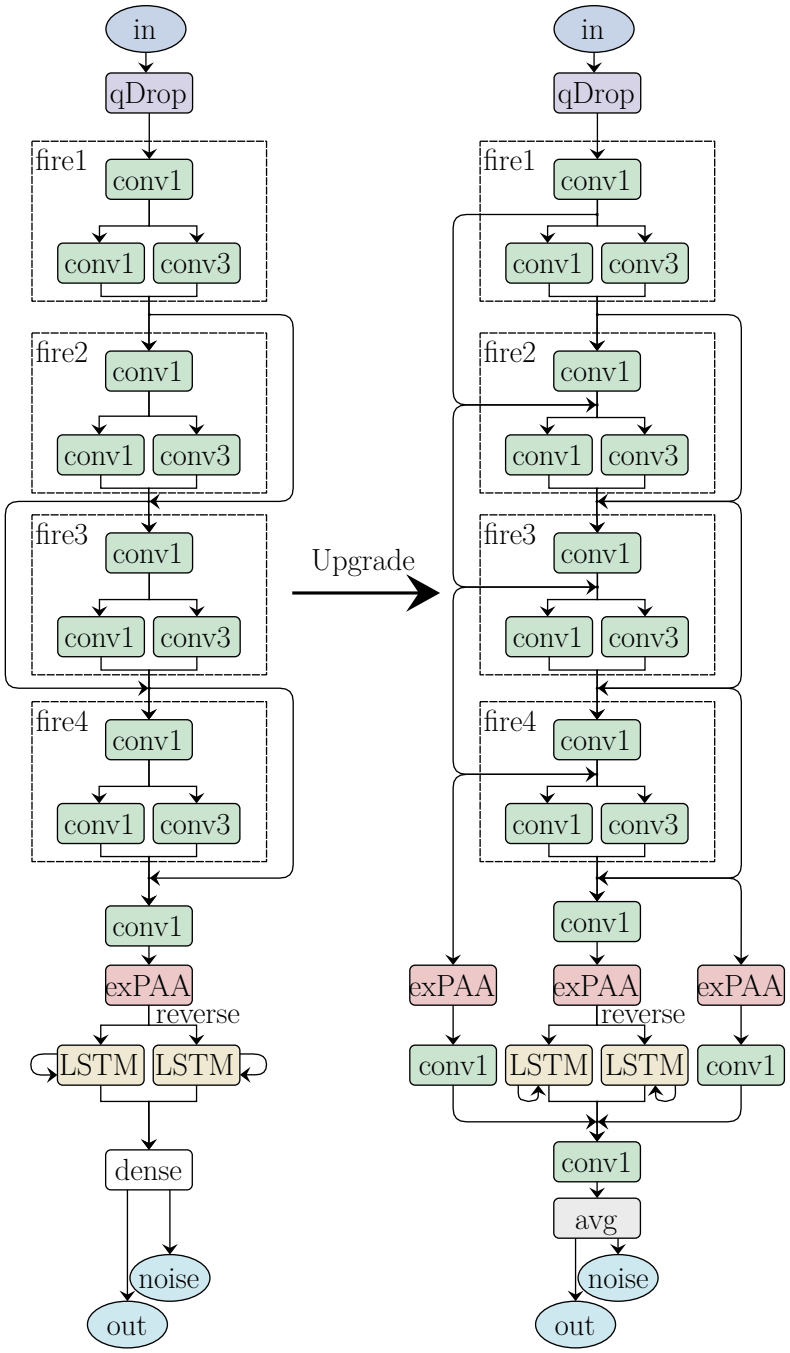


Figure 7.11: The macroarchitectural view of MarineNet (left) and upgraded MarineNet (right).

weights is reduced from $1\,145\,072$ to $376\,188$, which is factor of 3.04 less. Compared to the bLSTM architecture, this upgraded architecture requires a factor of 7.2 less weights. The general dropout probability is set to $p = .5$. All architectural changes are shown on the right side of Figure 7.11.

7.5.2 Training of Accurate Uncertainty Predictions

The original training of the dynamic noise function happens simultaneously with the point prediction. This has the downside that the actual scaling of this function has to be determined by manually tuning the trade-off parameter α.

To replace this hyper-parameter α and let the network optimizer determine the scaling of the noise function, I propose following changes: First, the network is trained without the dynamic noise function g by minimizing the MSE loss for the point prediction function f. Then, all weights are frozen, which means they can no longer be changed by the optimizer. The exception is the output layer for the noise function, which is now the only trainable layer. Now, I only train the dynamic noise layer by minimizing a new loss function:

$$
\begin{aligned}
L_{unc} := \quad & 2 \cdot \max\left(\quad \Lambda\left(\tfrac{51}{100}\right) \cdot \left(\tfrac{51}{100} - \mathrm{acc}\left(\tfrac{51}{100}\right)\right)^2, \quad 0 \right) \\
+ \quad & \sum_{i=53}^{97} \left| \quad \Lambda\left(\tfrac{i}{100}\right) \cdot \left(\tfrac{i}{100} - \mathrm{acc}\left(\tfrac{i}{100}\right)\right)^2 \quad \right| , \\
+ \quad & 2 \cdot \max\left(- \Lambda\left(\tfrac{99}{100}\right) \cdot \left(\tfrac{99}{100} - \mathrm{acc}\left(\tfrac{99}{100}\right)\right)^2, \quad 0 \right)
\end{aligned}
\tag{7.6}
$$

$\Lambda(i)$ defines the difference to the absolute prediction error and the uncertainty interval at i percent accuracy:

$$
\Lambda(i) := \left|\mathbb{E}[\boldsymbol{y}] - \boldsymbol{y}\right| - \sqrt{\widetilde{\mathrm{Var}}[\boldsymbol{y}]} \cdot \sqrt{2} \cdot \mathrm{erf}^{-1}(i) \ , \tag{7.7}
$$

with inverse Gauss error function erf^{-1}, ensemble predictive mean $\mathbb{E}[\boldsymbol{y}]$, ensemble variance $\widetilde{\mathrm{Var}}[\boldsymbol{y}]$, and accuracy $\mathrm{acc}(i)$ (Equation 7.4). Since the network only trains the noise output layer, the acquisition of the ensemble mean and variance by running the network multiple times without adjust-

ing the weights is not computationally expensive.

Without $\mathsf{acc}(i)$, the optimization would favor a calibration to 75% accuracy because the actual accuracies are not taken into account and the network finds its optimum between 51% and 99%. In the here used accuracy function, the number of patterns n are the batch size m ($n := m$). The $\mathsf{acc}(i)$ function cannot be used alone, since it is not differentiable, because of the logical operator ($\leq$).

The first and third row of Equation 7.6 can be seen as bounds, since they only increase in value when they fall below or exceed their respective accuracy of 51% and 99%. This is the same accuracy range that I consider for the Brier score (see Section 7.4.1). To increase the importance of these bounds, I doubled their output. The middle row contains points to support a good fit to individual accuracies between the bounds.

7.5.3 Experimental Evaluation

I conduct an experiment to compare the upgraded MarineNet to the original one, analyze the impact of parts from the time series on the prediction, and evaluate the automatically learned dynamic noise function. The same splitting of training and test data as before is utilized. Optimization is also the same with the removal of the trade-off parameter α and the addition of the qDrop exponent ε to the set of hyper-parameters. See Table 7.3 for exPAA and ε settings. I repeat the final model runs 30 times.

Again, if the one-sided Mann–Whitney U test results show a p-value below 0.05 and the U value is below or equal the critical value, I call it *significant*. The critical U value for 40 runs with MarineNet and 30 runs with the upgraded MarineNet, a standard score z of 1.64, and a significance level of 0.05, is 600.

In Figure 7.12 the best runs of MarineNet are compared to the upgraded models in box plots. The first row of plots shows the RMSE, which is significantly lower for *Temp*, *Conductivity*, *Pressure*, and *Direction*. I also found a significantly lower RMSE score for *Direction* compared to the bLSTM architecture from the previous chapter, when the original MarineNet did not improve. For the Brier score, the results of *Temp*, *Conductivity*, and *Pressure* are better, while there is no distinctable difference for *Direction*. A significantly lower standard uncertainty interval for *Pressure* and *Speed*

Table 7.3: Choice of optimized parameter settings for the upgraded MarineNet.

Sensor	#Steps δ	#Parts δ'	Exponent e	Quality exp.ε
Speed	72	4	2.0	.25
Temp	36	4	2.0	.0625
Conductivity	18	8	2.0	.03125
Pressure	36	8	1.5	.25
Direction	72	8	1.5	.25

is good for the former since the Brier Score is also lower, but the interval for *Speed* is too small according to the worse Brier score. The *Speed* models do not show better Brier score or RMSE results and the statistics will not allow further comparison (exceeded critical U value). The results for the linear calibrations show similar results to previous experiments; linear scaling is only significantly better for *Pressure*, the other results are equal or worse than the unscaled interval. To summarize, all except the *Speed* sensor model benefit from the changes made to the MarineNet architecture and training process.

With Figure 7.13, I demonstrate the added interpretability coming from the changed architecture in letter value plots [61]. It shows the activation distribution across the parts for *Speed*, *Conductivity*, and *Direction*. One entry to the distribution consists of the average output from the 512 filters per part. I uniformly sampled patterns from the training and test set to create the shown figure. The boxes define the observed quantile areas, the widest box covers the quartiles between 25% and 75%. Note that these parts do contain a different amount of real time steps, since the exPAA layer is applied before. For the *Speed* activations, which cover three hours in four parts, the importance increases with later parts. The *Conductivity* activations behave differently in their three hour time frame with eight parts, the first and the last four parts have relatively high impact. In the twelve hour frame with eight parts of the *Direction* model, the most important parts are the last three, although the last one has lower impact than the two previous ones. In effect, this visualization helps to understand the impact of individual parts. Often the last parts are most important, but other parts hold valuable information as well.

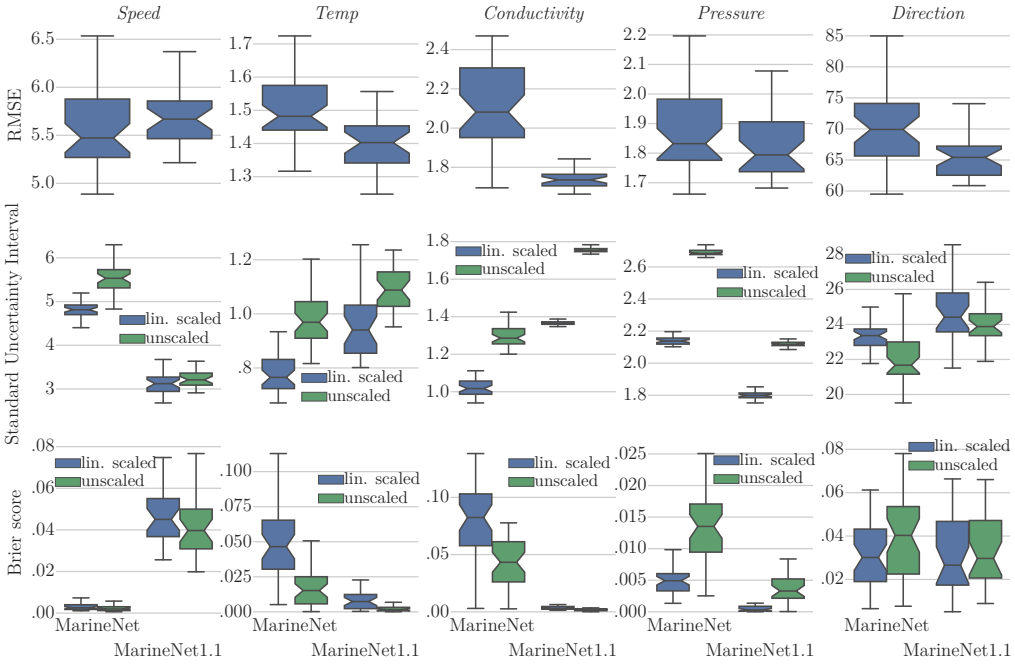


Figure 7.12: Comparing performance of MarineNet and upgraded MarineNet with box plots. The columns show the target sensors and the rows different quality measurements. MarineNet parameter settings with lowest RMSE are chosen for comparison.

7.6 Conclusion

With the MarineNet architecture, I improved and added to the previous bLSTM architecture. First, I included convolutional modules that filter the time series and showed that the used input sensors are often linearly, but also non-linearly correlated. Second, the model certainty is now available as part of the network output with uncertainty predictions. These innovations help to improve on the results from the previous chapter with a RMSE up to 52.16% lower than before. The input quality based dropout with qDrop offers more improvements to RMSE and reliable model uncertainty predictions. Beside the lower error rate, my models require 2.365 times less trainable weights and 66.66% fewer recursions than the previous models. This allows for more training in less iterations and again shows that the right network architecture can have a greater impact than

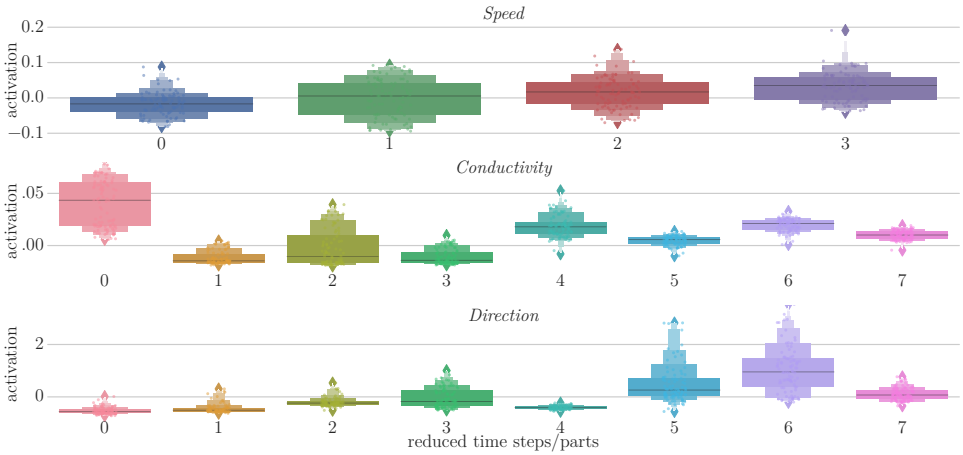


Figure 7.13: Impact of different parts on prediction presented in letter value plots [61]. Mean activation output distributions of last conv1 layer (y-axis) for three of the sensors. The x-axis shows the parts that each contain a different amount of time steps due to different exPAA settings.

the amount of used weights. Nevertheless, the tuning of the trade-off parameter α for the uncertainty prediction is difficult and requires many optimization runs.

By introducing a new loss function and a change to the training process, I am able to eliminate the need to find the trade-off parameter α. Further, through a small upgrade to the architecture by adding more shortcuts as well as replacing the dense layer with another conv layer, I have created better and smaller models for four of the five sensors. The highest improvement in RMSE is achieved by *Conductivity* with a 16.61% lower error compared to the first MarineNet. These changes provide insights into which part of the time series contributes the most to the networks output.

Although the architecture is created to solve the virtual sensor task of my marine dataset, it is flexible enough to be applied to other time series tasks. For example, the input quality information does not have to be imputation quality, other quality information can be used as well. The number of fire-modules has to be adjusted for the time resolution and the kind of features a practitioner wants to find.

Part III

Conclusion and Outlook

In this last part, I summarize the contributions of this work and outline limitations as well as possibilities for future research.

8 Conclusion

8.1 Summary

The ever growing amount of data is a prevailing trend in all applications. In the marine observation domain, a growing infrastructure of sensors measures data that is unprocessable manually. Moreover, although these sensors are robust, they eventually break due to the challenging environment. In this thesis, I focused on creating models to replace a broken flow sensor based on its surrounding sensors. These models build upon machine learning, the automatic learning of models from data without extensive domain knowledge. A machine learning model can perform a variety of tasks, making this field nearly universally applicable. In the following, I summarize the contributions of my thesis.

8.1.1 Part I: Dataset and Preprocessing

Part I focuses on the description and preprocessing of the data. First, I introduce the data from the TSS and BEFmate project in detail. This marine domain is challenging due to the dynamics created by the tides and seasonal influences. Ideally, neural networks need no preprocessing, but

some adjustments and modifications are necessary. In particular, besides general preprocessing, I correct drifts with statistical methods for sensors afflicted by biofouling.

As next step, I detect extreme events in the TSS data, which are evaluated by an expert round. To that end, I apply LOF with different feature handling methods to acquire an outlier score for each time step. The extreme events are then chosen with a top-k approach called top-k-time that reduces the detection of the same events within certain periods. A high true-positive rate underlines the usefulness of this extreme event detection. Moreover, a new visualization based on four reduced features calculated by ISOMAP is introduced.

The last contribution of this part is the penalized DTWkNN ensemble with a linear interpolation step, a novel approach to fill longer gaps in data. I evaluate this aproach in a comparison to state-of-the-art methods on 16 different and unrelated datasets. The results show that other machine learning approaches suffer from under-fitting as the damaged training dataset misses more values. In addition, my ensemble aproach has prevailed on most datasets when values for longer periods are missing.

8.1.2 Part II: Virtual Sensors with Deep Learning

In Part II, I create a virtual sensor of the dataset from the previous part based on deep learning techniques. To understand these techniques, I first introduce the basic as well as advanced concepts of deep learning used throughout this work. In recent years, the computational power became available to create deep neural networks, which led to a revival of machine learning with neural networks. This also accelerates the research in this area and new concepts emerged, such as dropout and batch normalization.

Thereafter, the first iteration of a deep learning architecture to model the failed flow sensor is presented. It gives important insights about the viability of the approach and how to further improve the architecture. The architecture is based on powerful albeit computationally expensive bLSTM layers. These layers learn from previous time steps, but in my marine application, the most recent steps hold the most information. To amplify the focus on this information, I introduce exPAA, a time dimensionality reduction method that retains fine details of recent values and coarser

details of more distant steps. This method yields good results as long as the assumption of declining information gain for older values holds true.

Finally, I propose various extensions for the architecture, which lead to the MarineNet architecture. These include the integration of convolutional layers to automatically learn features of higher abstraction. Further, predictive uncertainty through dropout MC predictions give insight about the reliability of the model. These additions show significant improvements over the first architecture. Then, the input quality is taken into account with a newly introduced input quality based dropout layer. In my case this quality is described by the number of consecutively imputed values. This supports the training phase of the models as they rely less on sensors that often fail and are able to compensate these failures with other sensors. An interesting finding is that it is better to impute the missing value and drop them in some MC runs than to ignore them completely. Then, based on the experiments results, I slightly changed the architecture and thus improved results while decreasing the number of weights for four of five sensor models. These changes also include the removal of a hyper-parameter by integrating it within the training process and observable impact of parts from a time series on the prediction. In summary, the created models based on the MarineNet architecture show good prediction results and provide valuable information about the predictive uncertainty.

8.2 Limitations and Outlook

Next, I will describe some of the limitations of this work. When appropriate, I will add ideas for future research.

A limitation that is always present in machine learning applications is the amount of data or rather the lack thereof. The widespread stigma that more available data corresponds to a better model, is true only to a certain extent. If the new data does not provide new information, they add no new value to the model. My virtual sensor model is trained on around 6.5 months of data. If the model were trained on the full dataset of about a year, it may generalize better. Nonetheless, more data from different seasonal events could create a better model. It would be interesting to evaluate the model with new data. One could consider unsupervised pretraining for the sensors that are available for longer periods, such as the

TSS sensors.

The extreme event detection is limited to my general definition of extreme events. Although each expert found one or more events that they are interested in, a specialization to an expert could produce better results. Further, the approach is unsupervised at the moment. The results on unexplored periods could also be improved through semi-supervised learning approaches, since a limited set of labeled events is now available.

The imputation quality of the DTWkNN ensemble depends on the window width with which the patterns are compared. This is a fixed setting and can be a limitation, since each dataset, feature, and missing reason requires a different value for the setting. Ideally, the window width setting would be flexible during the imputation process. One could cluster missing values beforehand and then optimize DTWkNN ensembles for each cluster, but this approach is computationally expensive. Future work could also explore other preprocessing steps, such as splines, since depending on the properties of the data, linear interpolation can be a poor choice.

The assumption of exPAA that recent values contain more relevant information than earlier ones may also be a limitation. Although I was able to improve my results with exPAA over normal PAA, dynamic parts could be useful as different or more specific assumptions could apply depending on the situation. But there is an open question regarding how to calculate the splits and when the calculations should be performed. It might be worth to study attention mechanisms [148] and apply them to exPAA without or only a small increase in the computational costs. Another approach might be to analyze the activations from the last conv layer of my updated architecture, because it contains information about the impact of individual time steps for a prediction.

At the moment, the MarineNet architecture is robust to temporary failure of sensors, but it does not consider permanent failures of any of the input sensors. One approach could be to always apply dropout to these missing sensors, which should still provide a useful prediction depending on the relation to the target sensor. Another approach could be to replace this sensor with another virtual sensor, although this is only reasonable as long as there are enough real sensors available. A more flexible approach, which could also integrate new sensors, could be to reinitialize the network, but also reuse the old weights via transfer-learning [151]. If a new sensor

is added, the number of neurons could also be increased. Exploring these approaches could benefit a system like this that will be modified due to maintenance or installation of new equipment.

A

TSS Sensors

Table A.1: Overview of used sensors of TSS from [120]. The specifications from the manufacturer are in italics.

Parameter	Dynamic range	Accuracy	Resolution	Stability of accuracy	Model	Manufacturer
Air						Lambrecht Meteorological Instruments
Temperature	−30...70°C	±0.1°C (at 0°C)	Not given		828	
Pressure	600...1,100 hPa	±0.3 hPa	Not given		8128	
Wind speed	0.7...50 m/s	±2% full scale	0.1 m/s		14576	
Wind direction	0...360°C	±1%	2.5°		14566	
Underwater					4H combination probe	-4H-JENA engineering
Temperature	−2...35°C	±0.035°C, *0.02° C*	Not given, *0.001° C*	1 year		
Electrical conductivity (inductive type)	1 – 65 mS/cm	±0.065 m/cm	Not given, *0.2mS/cm*[a] *0.001 mS/cm*	Nov-March: 3 months; April-Oct < 2 weeks		
Pressure	0 – 3,000 hPa	±3 hPa	Not given, *0.001 hPa*	1 year		

[a] Up to 20 mS/cm due to biofouling in periods of high biological productivity

B

Implementation

The main programming language used is `Python 3`. For my DTW implementation `Cython` [13] is employed for a fast runtime. Otherwise, all implementations are based on one or more of the following frameworks:

- `SciPy` [73]
  - `NumPy` [143]
  - `IPython` [116]
  - `Matplotlib` [65]
  - `pandas` [97]
  - `scikit-learn` [114]
- `TensorFlow` [1]
- `Keras` [26]
- `seaborn` [144]

I thank all open source contributors for their work.

List of Publications

Oehmcke, S., Zielinski, O., and Kramer, O. (2015). Event detection in marine time series data. In *Advances in Artificial Intelligence - Annual German Conference on AI (KI)*, pages 279–286. Springer

Oehmcke, S., Zielinski, O., and Kramer, O. (2016). kNN ensembles with penalized DTW for multivariate time series imputation. In *International Joint Conference on Neural Networks (IJCNN)*, pages 2774–2781. IEEE

Oehmcke, S., Zielinski, O., and Kramer, O. (2017b). Recurrent neural networks and exponential PAA for virtual marine sensors. In *International Joint Conference on Neural Networks (IJCNN)*, pages 4459–4466. IEEE

Oehmcke, S., Zielinski, O., and Kramer, O. (2017a). Input quality aware convolutional LSTM networks for virtual marine sensors. *Neurocomputing*, 275:2603–2615

List of Figures

List of Tables

Acronyms

BEFmate Biodiversity-Ecosystem Functioning across marine and terrestrial ecosystems. i, ii, 2, 5, 9–13, 15, 16, 19, 34, 35, 57, 61, 63, 83, 84, 90, 95, 106, 127, 139

bLSTM bidirectional long short-term memory network. ii, 6, 75, 76, 83, 89–91, 94, 95, 97, 98, 102, 103, 105, 106, 109, 114, 117, 119, 120, 122, 128

CNN convolutional neural network. ii, 66, 71, 75, 97

DTW dynamic time warping. i, ii, 33–35, 42–52, 54–58, 60, 61, 86, 98, 128, 130, 135, 167

exPAA exponential piecewise approximate aggregation. ii, 83, 84, 86, 88–95, 97, 102, 105, 108, 114, 117, 120, 121, 123, 128, 130

GPU graphics processing unit. 65, 86

ISOMAP isometric mapping. 22, 29–32, 128

***k*NN** *k*-nearest neighbors algorithm. i, ii, 33–36, 41, 42, 44–46, 48–52, 54–58, 60, 61, 65, 98, 99, 128, 130, 167

LOF local outlier factor. 22, 24, 25, 28, 29, 32, 128

LSTM long short-term memory network. 66, 73–76, 80, 83, 86, 87, 94, 97, 105

MC Monte Carlo. 81, 82, 105

MSE mean squared error. 67, 119

PAA piecewise approximate aggregation. 84, 87, 88, 93, 95, 130

qDrop input quality based dropout. 98–100, 102, 103, 106, 111, 112, 114–116, 120, 122

ReLU rectified linear unit. 72, 89, 105

RMedSE root median squared error. 91–94

RMSE root mean squared error. 91–94, 106, 107, 109–115, 120–123

RNN recurrent neural network. 66, 73–76

SGD stochastic gradient descent. 69

sigmoid logistic curve. 73, 74, 89, 105

tanh hyperbolic tangent. 73, 74, 89, 105

TSS Time Series Station Spiekeroog. i, ii, 2, 5, 9–12, 15–17, 19, 21, 22, 24, 27, 32–35, 39, 46, 53, 56–58, 61, 63, 84, 90, 95, 106, 114, 127, 128, 130, 134, 139

Nomenclature

$\boldsymbol{p}$	penalty vector	m	batch size
$\boldsymbol{p}^{quality}$	quality vector	n^{MC}	number of Monte Carlo runs
$\boldsymbol{X}$	label/target space	d'	number of neurons/kernel
$\boldsymbol{Y}$	label/target space	n	number of patterns/time steps
δ'	reduced window/period length	X	input matrix
Δ	distance measure	$\boldsymbol{x}$	pattern/input vector
δ	window/period length	Y	output matrix
$\hat{\boldsymbol{y}}$	predicted target vector	$\boldsymbol{y}$	true target vector
M	missing mask matrix	d	dimensionality
$\boldsymbol{m}$	missing mask vector	$N_k(\cdot)$	k nearest neighbors function
F	model space	$o_{score}(\cdot)$	outlier score function
f	model	P	penalty matrix

Notation

In this work, I use the following notation.

- Scalar values are denote as lower case letters (e.g. x);
- Vectors as bold lower case letters (e.g. $\boldsymbol{x}$);
- Matrices as upper case letters (e.g. X).

The subscript t of a vector $\boldsymbol{x}_t$ references to its temporal position in the training set ($t \in (1, \ldots, n)$). The subscript i of a vector x_i references to a scalar at that index. For a matrix $X_{i,j}$, the subscripts i and j refer to the row and column elements, respectively. Further, a matrix $X_{i:i',j}$ returns a vector from row i to i' of column j. A matrix is returned, if both subscript indices are ranges (e.g. $X_{i:i',j:j'}$). If a matrix $X_{i:,:j}$ is given, a matrix from row i to the last row and from the first column to column j is returned.

References

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *Operating Systems Design and Implementation (OSDI)*, volume 16, pages 265–283.

[2] Alippi, C., Boracchi, G., and Roveri, M. (2012). On-line reconstruction of missing data in sensor/actuator networks by exploiting temporal and spatial redundancy. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

[3] Andreas, J., Rohrbach, M., Darrell, T., and Klein, D. (2016). Neural module networks. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 39–48. IEEE.

[4] Atkinson, C., Long, T. W., and Hanzevack, E. L. (1998). Virtual sensing: A neural network-based intelligent performance and emissions prediction system for on-board diagnostics and engine control. Technical report, SAE Technical Paper.

[5] Auslander, B., Gupta, K. M., and Aha, D. W. (2011). A comparative evaluation of anomaly detection algorithms for maritime video surveillance. In *Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense*, pages 1–14. SPIE.

[6] Badewien, T. H., Zimmer, E., Bartholomä, A., and Reuter, R. (2009). Towards continuous long-term measurements of suspended

particulate matter (spm) in turbid coastal waters. *Ocean Dynamics*, 59(2):227–238.

[7] Baladrón, C., Aguiar, J. M., Calavia, L., Carro, B., Sánchez-Esguevillas, A., and Hernández, L. (2012). Performance study of the application of artificial neural networks to the completion and prediction of data retrieved by underwater sensors. *Sensors*, 12(2):1468–1481.

[8] Balke, T., Lõhmus, K., Hillebrand, H., Zielinski, O., Haynert, K., Meier, D., Hodapp, D., Minden, V., and Kleyer, M. (2017). Experimental salt marsh islands: A model system for novel metacommunity experiments. *Estuarine, Coastal and Shelf Science*, 198, Part A:288–298.

[9] Balke, T., Stock, M., Jensen, K., Bouma, T. J., and Kleyer, M. (2016). A global analysis of the seaward salt marsh extent: The importance of tidal range. *Water Resources Research*, 52(5):3775–3786.

[10] Basu, S. and Meckesheimer, M. (2007). Automatic outlier detection for time series: an application to sensor data. *Knowledge and Information Systems (KAIS)*, 11(2):137–154.

[11] Batista, G. E. A. P. A. and Monard, M. C. (2003). An analysis of four missing data treatment methods for supervised learning. *Applied Artificial Intelligence (AAI)*, 17(5-6):519–533.

[12] Baydin, A. G., Pearlmutter, B. A., and Radul, A. A. (2015). Automatic differentiation in machine learning: a survey. *Computing Research Repository (CoRR)*, abs/1502.05767.

[13] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2011). Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39.

[14] Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*, pages 437–478. Springer.

[15] Bishop, C. M. (2006). *Pattern recognition and machine learning*, volume 1. Springer.

[16] Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. K. (1987). Occam's razor. *Information processing letters*, 24(6):377–380.

[17] Box, G. E. and Cox, D. R. (1964). An analysis of transformations. *Journal of the Royal Statistical Society. Series B - Methodological*, pages 211–252.

[18] Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2):123–140.

[19] Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

[20] Breunig, M. M., Kriegel, H., Ng, R. T., and Sander, J. (2000). Lof: Identifying density-based local outliers. In *SIGMOD International Conference on Management of Data (SIGMOD/PODS)*, SIGMOD '00, pages 93–104. ACM.

[21] Brier, G. W. (1950). Verification of forecasts expressed in terms of probability. *Monthey Weather Review*, 78(1):1–3.

[22] Chakrabarti, K., Keogh, E. J., Mehrotra, S., and Pazzani, M. J. (2002). Locally adaptive dimensionality reduction for indexing large

time series databases. *Transactions on Database Systems (TODS)*, 27(2):188–228.

[23] Chambers, J. M., Cleveland, W. S., Kleiner, B., Tukey, P. A., et al. (1983). *Graphical methods for data analysis*, volume 5. Wadsworth Belmont, CA.

[24] Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *Computing Surveys (CSUR)*, 41(3):15:1–15:58.

[25] Cho, K., van Merrienboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. In *Workshop on Syntax, Semantics and Structure in Statistical Translation (SSST)*.

[26] Chollet, F. et al. (2015). Keras. `https://github.com/fchollet/keras`.

[27] Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: deep neural networks with multitask learning. In *International Conference on Machine Learning (ICML)*, pages 160–167. International Machine Learning Society.

[28] Dalto, M., Matuško, J., and Vašak, M. (2015). Deep neural networks for ultra-short-term wind forecasting. In *International Conference on Industrial Technology (ICIT)*, pages 1657–1663. IEEE.

[29] Denk, M. and Weber, M. (2011). Avoid filling Swiss cheese with whipped cream: imputation techniques and evaluation procedures for cross-country time series. *IMF Working Papers*, pages 1–27.

[30] Díaz, D., Torres, A., and Dorronsoro, J. R. (2015). Deep neural networks for wind energy prediction. In Rojas, I. and Joya, Gonzalo band Catala, A., editors, *International Work-Conference on Artificial Neural Networks (IWANN)*, pages 430–443. Springer.

[31] Dietterich, T. G. (2000). Ensemble methods in machine learning. In Kittler, J. and Roli, F., editors, *Multiple Classifier Systems*, volume 1857 of *Lecture Notes in Computer Science*, pages 1–15. Springer.

[32] Ding, H., Trajcevski, G., Scheuermann, P., Wang, X., and Keogh, E. J. (2008). Querying and mining of time series data: experimental comparison of representations and distance measures. *Proceedings of the VLDB Endowment (PVLDB)*, 1:1542–1552.

[33] Doerffer, R., Colijn, F., and Beusekom, J. v. (2008). Observing the coastal sea-an atlas of advanced monitoring techniques.

[34] Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87.

[35] Dugas, C., Bengio, Y., Bélisle, F., Nadeau, C., and Garcia, R. (2000). Incorporating second-order functional knowledge for better option pricing. In *Advances in Neural Information Processing Systems: Annual Conference on Neural Information Processing Systems (NIPS)*, pages 472–478. MIT.

[36] Dyke, P. (2016). *Modelling Coastal and Marine Processes*. World Scientific Publishing Co Inc.

[37] Eck, D. and Schmidhuber, J. (2002). Finding temporal structure in music: blues improvisation with LSTM recurrent networks. In *Workshop on Neural Networks for Signal Processing (NNSP)*, pages 747–756. IEEE.

[38] Filonov, P., Lavrentyev, A., and Vorontsov, A. (2016). Multivariate industrial time series with cyber-attack simulation: Fault detection using an lstm-based predictive data model. *Computing Research Repository (CoRR)*, abs/1612.06676.

[39] Fleiss, J. L. (1971). Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378.

[40] Fletcher, R. (1987). Practical methods of optimization john wiley & sons. *New York*, 80.

[41] Fritsch, F. N. and Carlson, R. E. (1980). Monotone piecewise cubic interpolation. *SIAM Journal on Numerical Analysis*, 17(2):238–246.

[42] Gal, Y. (2016). *Uncertainty in deep learning*. PhD thesis, University of Cambridge.

[43] Gal, Y. and Ghahramani, Z. (2016). A theoretically grounded application of dropout in recurrent neural networks. In Lee, D. D., Sugiyama, M., von Luxburg, U., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems: Annual Conference on Neural Information Processing Systems (NIPS)*, pages 1019–1027.

[44] Garaba, S., Badewien, T., Braun, A., Schulz, A.-C., and Zielinski, O. (2014). Using ocean colour remote sensing products to estimate turbidity at the wadden sea time series station spiekeroog. *Journal of the European Optical Society - Rapid publications*, 9(0).

[45] García, S., Luengo, J., and Herrera, F. (2015). Dealing with missing values. In *Data Preprocessing in Data Mining*, volume 72 of *Intelligent Systems Reference Library*, pages 59–105. Springer International Publishing.

[46] Gers, F. A., Schmidhuber, J., and Cummins, F. A. (2000). Learning to forget: Continual prediction with lstm. *Neural Computation*, 12(10):2451–2471.

[47] Gill, P. E., Murray, W., and Wright, M. H. (1981). Practical optimization.

[48] Girshick, R. B., Donahue, J., Darrell, T., and Malik, J. (2016). Region-based convolutional networks for accurate object detection and segmentation. *Transactions on Pattern Analysis and Machine Intelligence*, 38:142–158.

[49] Giusti, A., Ciresan, D. C., Masci, J., Gambardella, L. M., and Schmidhuber, J. (2013). Fast image scanning with deep max-pooling convolutional neural networks. *International Conference on Image Processing (ICIP)*, pages 4034–4038.

[50] Gonzaga, J., Meleiro, L. A. C., Kiang, C., and Maciel Filho, R. (2009). Ann-based soft-sensor for real-time process monitoring and control of an industrial polymerization process. *Computers & Chemical Engineering*, 33(1):43–49.

[51] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT.

[52] Graves, A., Fernández, S., and Schmidhuber, J. (2005). Bidirectional LSTM networks for improved phoneme classification and recognition. In *Artificial Neural Networks: Formal Models and Their Applications (ICANN)*, pages 799–804. Springer.

[53] Graves, A., rahman Mohamed, A., and Hinton, G. E. (2013). Speech recognition with deep recurrent neural networks. *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6645–6649.

[54] Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2015). Lstm: A search space odyssey. *Computing Research Repository (CoRR)*, abs/1503.04069.

[55] Gupta, M., Gao, J., Aggarwal, C. C., and Han, J. (2014a). Outlier detection for temporal data. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 5(1):1–129.

[56] Gupta, V., Kenny, P., Ouellet, P., and Stafylakis, T. (2014b). I-vector-based speaker adaptation of deep neural networks for french broadcast audio transcription. *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6334–6338.

[57] Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The elements of statistical learning*, volume 2. Springer.

[58] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778. IEEE.

[59] He, Q. P. and Wang, J. (2007). Fault detection using the k-nearest neighbor rule for semiconductor manufacturing processes. *Transactions on Semiconductor Manufacturing*, 20(4):345–354.

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

[61] Hofmann, H., Kafadar, K., and Wickham, H. (2011). Letter-value plots: Boxplots for large data. Technical report, had.co.nz.

[62] Hsu, H., Yang, A. C., and Lu, M. (2011). KNN-DTW based missing value imputation for microarray time series data. *Journal of Computers (JCP)*, 6(3):418–425.

[63] Huang, G., Liu, Z., van der Maaten, L., and Weinberger, K. Q. (2017). Densely connected convolutional networks. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269. IEEE.

[64] Huang, J. and Kingsbury, B. (2013). Audio-visual deep learning for noise robust speech recognition. *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7596–7599.

[65] Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95.

[66] Hunter, J. S. (1986). The exponentially weighted moving average. *Journal of Quality Technology (JQT)*, 18(4):203–210.

[67] Hyndman, R. J. and Athanasopoulos, G. (2014). *Forecasting: Principles and practice*. OTexts.

[68] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. (2016). Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. *Computing Research Repository (CoRR)*, abs/1602.07360.

[69] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning (ICML)*, pages 448–456. International Machine Learning Society.

[70] Jahnke, J. A. (2000). *Continuous emission monitoring*. John Wiley & Sons.

[71] Jerez, J. M., Molina, I., García-Laencina, P. J., Alba, E., Ribelles, N., Martín, M., and Franco, L. (2010). Missing data imputation using statistical and machine learning methods in a real breast cancer problem. *Artificial Intelligence in Medicine*, 50(2):105–115.

[72] Jin, W., Tung, A. K., and Han, J. (2001). Mining top-n local outliers in large databases. In *International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 293–298. ACM.

[73] Jones, E., Oliphant, T., Peterson, P., et al. (2001). SciPy: Open source scientific tools for Python.

[74] Jr., J. R. B., do Carmo Nicoletti, M., and Zhao, L. (2014). Imputation of missing data supported by complete p-partite attribute-based decision graphs. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1100–1106. IEEE.

[75] Kadlec, P., Gabrys, B., and Strandt, S. (2009). Data-driven soft sensors in the process industry. *Computers & Chemical Engineering*, 33(4):795–814.

[76] Keogh, E., Chakrabarti, K., Pazzani, M., and Mehrotra, S. (2001). Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and information Systems (KAIS)*, 3(3):263–286.

[77] Keogh, E. and Folias, T. (2002). The UCR time series data mining archive.

[78] Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. (2017). On large-batch training for deep learning: Generalization gap and sharp minima. In *International Conference on Learning Representations (ICLR)*.

[79] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *Computing Research Repository (CoRR)*, abs/1412.6980.

[80] Kramer, O., Gieseke, F., and Satzger, B. (2013). Wind energy prediction and monitoring with neural computation. *Neurocomputing*, 109:84–93.

[81] Kriegel, H.-P., Kröger, P., Schubert, E., and Zimek, A. (2011). Interpreting and unifying outlier scores. In *SIAM International Conference on Data Mining (SDM)*, pages 13–24. SIAM.

[82] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems: Annual Conference on Neural Information Processing Systems (NIPS)*, pages 1106–1114. Curran Associates.

[83] Kruskal, J. B. and Liberman, M. (1983). The symmetric time-warping problem: from continuous to discrete. *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, pages 125–161.

[84] Landis, J. R. and Koch, G. G. (1977). The measurement of observer agreement for categorical data. *Biometrics*, pages 159–174.

[85] LeCun, Y., Bottou, L., Orr, G. B., and Müller, K.-R. (1998). Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–50. Springer.

[86] Leong, C. C., Blakey, S., and Wilson, C. W. (2016). Genetic algorithm optimised chemical reactors network: A novel technique for alternative fuels emission prediction. *Swarm and Evolutionary Computation*, 27:180–187.

[87] Li, X., Han, J., Kim, S., and Gonzalez, H. (2007). Roam: Rule-and motif-based anomaly detection in massive moving object data sets. In *SIAM International Conference on Data Mining (SDM)*, volume 7, pages 273–284. SIAM.

[88] Liang, M. and Hu, X. (2015). Recurrent convolutional neural network for object recognition. *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3367–3375.

[89] Liang, M., Hu, X., and Zhang, B. (2015). Convolutional neural networks with intra-layer recurrent connections for scene labeling. In

Advances in Neural Information Processing Systems: Annual Conference on Neural Information Processing Systems (NIPS). Curran Associates.

[90] Lin, M., Chen, Q., and Yan, S. (2013). Network in network. *Computing Research Repository (CoRR)*.

[91] Liu, F. T., Ting, K. M., and Zhou, Z.-H. (2008). Isolation forest. In *International Conference on Data Mining (ICDM)*, pages 413–422. IEEE.

[92] Maduranga, D., Zheng, J., Mundra, P. A., and Rajapakse, J. C. (2013). Inferring gene regulatory networks from time-series expressions using random forests ensemble. In *International Conference on Pattern Recognition in Bioinformatics (IAPR)*, pages 13–22. Springer.

[93] Malone, T. C. (2003). The coastal module of the global ocean observing system (goos): an assessment of current capabilities to detect change. *Marine Policy*, 27(4):295–302.

[94] Manevitz, L. M., Bitar, A., and Givoli, D. (2005). Neural network time series forecasting of finite-element mesh adaptation. *Neurocomputing*, 63:447–463.

[95] Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, pages 50–60.

[96] Maronna, R., Martin, D., and Yohai, V. (2006). *Robust statistics*. John Wiley & Sons, Chichester. ISBN.

[97] McKinney, W. (2010). Data structures for statistical computing in python. In van der Walt, S. and Millman, J., editors, *Proceedings of the 9th Python in Science Conference*, pages 51–56.

[98] McKnight, P. E. and Najab, J. (2010). *Mann-Whitney U Test*. John Wiley & Sons, Inc.

[99] Modenesi, A. P. and Braga, A. P. (2009). Analysis of time series novelty detection strategies for synthetic and real data. *Neural Processing Letters*, 30(1):1–17.

[100] Mohri, M., Rostamizadeh, A., and Talwalkar, A. (2012). *Foundations of machine learning*. Adaptive computation and machine learning. MIT.

[101] Morales, F. J. O. and Roggen, D. (2016). Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition. In *Sensors*, volume 16, page 115. Molecular Diversity Preservation International.

[102] Nagi, J., Ducatelle, F., Caro, G. A. D., Ciresan, D. C., Meier, U., Giusti, A., Nagi, F., Schmidhuber, J., and Gambardella, L. M. (2011). Max-pooling convolutional neural networks for vision-based hand gesture recognition. *International Conference on Signal and Image Processing Applications (ICSIPA)*, pages 342–347.

[103] Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *International Conference on Machine Learning (ICML)*. International Machine Learning Society.

[104] Ning, Y. and Chen, X. (2009). Coal face gas emission prediction based on support vector machine. In *International Conference on Artificial Intelligence and Computational Intelligence (AICI)*, volume 1, pages 19–22. IEEE.

[105] Oehmcke, S., Zielinski, O., and Kramer, O. (2015). Event detection in marine time series data. In *Advances in Artificial Intelligence - Annual German Conference on AI (KI)*, pages 279–286. Springer.

[106] Oehmcke, S., Zielinski, O., and Kramer, O. (2016). kNN ensembles with penalized DTW for multivariate time series imputation. In *International Joint Conference on Neural Networks (IJCNN)*, pages 2774–2781. IEEE.

[107] Oehmcke, S., Zielinski, O., and Kramer, O. (2017a). Input quality aware convolutional LSTM networks for virtual marine sensors. *Neurocomputing*, 275:2603–2615.

[108] Oehmcke, S., Zielinski, O., and Kramer, O. (2017b). Recurrent neural networks and exponential PAA for virtual marine sensors. In *International Joint Conference on Neural Networks (IJCNN)*, pages 4459–4466. IEEE.

[109] Palomares, M. L. D. and Pauly, D. (1998). Predicting food consumption of fish populations as functions of mortality, food type, morphometrics, temperature and salinity. *Marine and Freshwater Research*, 49(5):447–453.

[110] Pan, L. and Li, J. (2010). K-nearest neighbor based missing data estimation algorithm in wireless sensor networks. *Wireless Sensor Network*, 2(2):115–122.

[111] Parry, R., Jones, W., Stokes, T., Phan, J., Moffitt, R., Fang, H., Shi, L., Oberthuer, A., Fischer, M., Tong, W., et al. (2010). k-nearest neighbor models for microarray gene expression analysis and clinical outcome prediction. *The pharmacogenomics journal*, 10(4):292–309.

[112] Paskin, M., Guestrin, C., and McFadden, J. (2005). A robust architecture for distributed inference in sensor networks. In *International Symposium on Information Processing in Sensor Networks (IPSN)*, volume 4, pages 55–62. IEEE.

[113] Pearson, K. (1920). Notes on the history of correlation. *Biometrika*, pages 25–45.

[114] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research (JLMR)*, 12:2825–2830.

[115] Pinheiro, P. H. O. and Collobert, R. (2014). Recurrent convolutional neural networks for scene labeling. In *International Conference on Machine Learning (ICML)*. International Machine Learning Society.

[116] Pérez, F. and Granger, B. E. (2007). Ipython: A system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29.

[117] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145–151.

[118] Rahman, S. A., Huang, Y., Claassen, J., and Kleinberg, S. (2014). Imputation of missing values in time series with lagged correlations. In *International Conference on Data Mining Workshops (ICDM)*, pages 753–762. IEEE.

[119] Ratanamahatana, C. A. and Keogh, E. J. (2005). Three myths about dynamic time warping data mining. In *SIAM International Conference on Data Mining (SDM)*, pages 506–510. SIAM.

[120] Reuter, R., Badewien, T. H., Bartholomä, A., Braun, A., Lübben, A., and Rullkötter, J. (2009). A hydrographic time series station in the Wadden Sea (southern North Sea). *Ocean Dynamics*, 59(2):195–211.

[121] Riveiro, M., Falkman, G., and Ziemke, T. (2008). Improving maritime anomaly detection and situation awareness through interactive visualization. In *International Conference on Information Fusion (FUSION)*, volume 11, pages 1–8. IEEE.

[122] Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton*. Cornell Aeronautical Laboratory.

[123] Roverso, D. (2009). Empirical ensemble-based virtual sensing-a novel approach to oil-in-water monitoring. In *Workshop on Oil-in-Water Monitoring (TÜV NEL)*.

[124] Rubin, D. B. (2004). *Multiple imputation for nonresponse in surveys*, volume 81. John Wiley & Sons.

[125] Ruder, S. (2016). An overview of gradient descent optimization algorithms. *Computing Research Repository (CoRR)*, abs/1609.04747.

[126] Sainath, T. N., Vinyals, O., Senior, A. W., and Sak, H. (2015). Convolutional, long short-term memory, fully connected deep neural networks. *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4580–4584.

[127] Schafer, J. L. (1997). *Analysis of incomplete multivariate data*. CRC.

[128] Schureman, P. (1958). *Manual of harmonic analysis and prediction of tides*. Number 98. US Government Printing Office.

[129] Schuster, M. and Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *Transactions on Signal Processing*, 45(11):2673–2681.

[130] Shi, X., Chen, Z., Wang, H., Yeung, D.-Y., Wong, W.-K., and chun Woo, W. (2015). Convolutional lstm network: A machine learning approach for precipitation nowcasting. In *Advances in Neural*

Information Processing Systems: Annual Conference on Neural Information Processing Systems (NIPS). Curran Associates.

[131] Shokoohi-Yekta, M., Wang, J., and Keogh, E. J. (2015). On the non-trivial generalization of dynamic time warping to the multi-dimensional case. In *SIAM International Conference on Data Mining (SDM)*.

[132] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *Computing Research Repository (CoRR)*, abs/1409.1556.

[133] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research (JLMR)*, 15(1):1929–1958.

[134] Subbaraj, P. and Kannapiran, B. (2010). Artificial neural network approach for fault detection in pneumatic valve in cooler water spray system. *International Journal of Computer Applications (IJCA)*, 9(7).

[135] Sundermeyer, M., Ney, H., and Schlüter, R. (2015). From feedforward to recurrent lstm neural networks for language modeling. *Transactions on Audio, Speech, and Language Processing*, 23:517–529.

[136] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9. IEEE.

[137] ten Holt, G. A., Reinders, M. J., and Hendriks, E. (2007). Multi-dimensional dynamic time warping for gesture recognition. In *Thir-*

teenth annual conference of the Advanced School for Computing and Imaging, volume 300.

[138] Tenenbaum, J. B., De Silva, V., and Langford, J. C. (2000). A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323.

[139] Thadathil, P., Bajish, C., Behera, S., and Gopalakrishna, V. (2012). Drift in salinity data from argo profiling floats in the sea of japan. *Journal of Atmospheric and Oceanic Technology*, 29(1):129–138.

[140] Tieleman, T. and Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31.

[141] Troyanskaya, O. G., Cantor, M. N., Sherlock, G., Brown, P. O., Hastie, T., Tibshirani, R., Botstein, D., and Altman, R. B. (2001). Missing value estimation methods for DNA microarrays. *Bioinformatics*, 17(6):520–525.

[142] Tsironi, E., Barros, P., Weber, C., and Wermter, S. (2017). An analysis of convolutional long short-term memory recurrent neural networks for gesture recognition. *Neurocomputing*, 268:76–86.

[143] van der Walt, S., Colbert, S. C., and Varoquaux, G. (2011). The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30.

[144] Waskom, M., Botvinnik, O., O'Kane, D., Hobson, P., Lukauskas, S., Gemperline, D. C., Augspurger, T., Halchenko, Y., Cole, J. B., Warmenhoven, J., de Ruiter, J., Pye, C., Hoyer, S., Vanderplas, J., Villalba, S., Kunter, G., Quintero, E., Bachant, P., Martin, M., Meyer, K., Miles, A., Ram, Y., Yarkoni, T., Williams, M. L., Evans, C.,

Fitzgerald, C., Brian, Fonnesbeck, C., Lee, A., and Qalieh, A. (2017). seaborn.

[145] Woon, W. L. and Kramer, O. (2016). Enhanced SVR ensembles for wind power prediction. In *International Joint Conference on Neural Networks (IJCNN)*, pages 2743–2748. IEEE.

[146] Woon, W. L., Oehmcke, S., and Kramer, O. (2017). Spatio-temporal wind power prediction using recurrent neural networks. In *International Conference in Neural Information Processing (ICONIP)*. Springer.

[147] Worzyk, N.-S. (2016). Erweiterungen für Penalty DTW*k*NN zur Datenimputation von maritimen Zeitreihen. Master's thesis, University Oldenburg.

[148] Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhudinov, R., Zemel, R., and Bengio, Y. (2015). Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning (ICML)*, pages 2048–2057. International Machine Learning Society.

[149] Xu, Y., Kong, Q., Huang, Q., Wang, W., and Plumbley, M. D. (2017). Convolutional gated recurrent neural network incorporating spatial features for audio tagging. *Computing Research Repository (CoRR)*, abs/1702.07787.

[150] Yi, S., Ju, J., Yoon, M.-K., and Choi, J. (2017). Grouped convolutional neural networks for multivariate time series. *Computing Research Repository (CoRR)*, abs/1703.09938.

[151] Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems: Annual Conference on Neural*

Information Processing Systems (NIPS), pages 3320–3328. Curran Associates.

[152] Yu, L., Wang, N., and Meng, X. (2005). Real-time forest fire detection with wireless sensor networks. *International Conference on Wireless Communications, Networking and Mobile Computing (WiMob)*, 2:1214–1217.

[153] Zhang, G. P. (2003). Time series forecasting using a hybrid arima and neural network model. *Neurocomputing*, 50:159–175.

[154] Zhang, K., Sun, M., Han, X., Yuan, X., Guo, L., and Liu, T. (2017). Residual networks of residual networks: multilevel residual networks. *Transactions on Circuits and Systems for Video Technology*.

[155] Zhao, R., Yan, R., Wang, J., and Mao, K. (2017). Learning to monitor machine health with convolutional bi-directional lstm networks. *Sensors*, 17(2):273.

[156] Zhou, J. and Tung, A. K. (2015). Smiler: A semi-lazy time series prediction system for sensors. In *SIGMOD International Conference on Management of Data (SIGMOD/PODS)*, pages 1871–1886. ACM.

[157] Zhou, X., Hu, B., Chen, Q., and Wang, X. (2017). Recurrent convolutional neural network for answer selection in community question answering. *Neurocomputing*, 274.

[158] Zielinski, O., Busch, J. A., Cembella, A. D., Daly, K. L., Engelbrektsson, J., Hannides, A. K., and Schmidt, H. (2009). Detecting marine hazardous substances and organisms: sensors for pollutants, toxins, and pathogens. *Ocean Science*, 5(3):329–349.