applied sciences

MDPI

Article

Online Machine Learning and Surrogate-Model-Based Optimization for Improved Production Processes Using a Cognitive Architecture

Alexander Hinterleitner [1], Richard Schulz [1], Lukas Hans [1], Aleksandr Subbotin [1], Nils Barthel [1], Noah Pütz [1], Martin Rosellen [1], Thomas Bartz-Beielstein [1,*], Christoph Geng [2] and Phillip Priss [2]

[1] Institute for Data Science, Engineering and Analytics, TH Köln University of Applied Sciences, 51643 Gummersbach, Germany; alexander.hinterleitner@th-koeln.de (A.H.); richard.schulz@th-koeln.de (R.S.); lukas.hans@th-koeln.de (L.H.); aleksandr.subbotin@th-koeln.de (A.S.); nils.barthel@th-koeln.de (N.B.); noah.c.puetz@gmail.com (N.P.); martin.rosellen@th-koeln.de (M.R.)

[2] Institute for Industrial Information Technology—inIT, OWL University of Applied Sciences and Arts, 32657 Lemgo, Germany; christoph.geng@th-owl.de (C.G.); phillip.priss@th-owl.de (P.P.)

* Correspondence: thomas.bartz-beielstein@th-koeln.de

Abstract: Cyber-Physical Systems (CPS) play an essential role in today's production processes, leveraging Artificial Intelligence (AI) to enhance operations such as optimization, anomaly detection, and predictive maintenance. This article reviews a cognitive architecture for Artificial Intelligence, which has been developed to establish a standard framework for integrating AI solutions into existing production processes. Given that machines in these processes continuously generate large streams of data, Online Machine Learning (OML) is identified as a crucial extension to the existing architecture. To substantiate this claim, real-world experiments using a slitting machine are conducted, to compare the performance of OML to traditional Batch Machine Learning. The assessment of contemporary OML algorithms using a real production system is a fundamental innovation in this research. The evaluations clearly indicate that OML adds significant value to CPS, and it is strongly recommended as an extension of related architectures, such as the cognitive architecture for AI discussed in this article. Additionally, surrogate-model-based optimization is employed, to determine the optimal hyperparameter settings for the corresponding OML algorithms, aiming to achieve peak performance in their respective tasks.

Keywords: machine learning; online algorithms; cyber-physical production systems; surrogate-based optimization

Citation: Hinterleitner, A.; Schulz, R.; Hans, L.; Subbotin, A.; Barthel, N.; Pütz, N.; Rosellen, M.; Bartz-Beielstein, T.; Geng, C.; Priss, P. Online Machine Learning and Surrogate-Model-Based Optimization for Improved Production Processes Using a Cognitive Architecture. *Appl. Sci.* **2023**, *13*, 11506. https://doi.org/10.3390/app132011506

Academic Editor: Michael Affenzeller

Received: 28 September 2023
Revised: 12 October 2023
Accepted: 13 October 2023
Published: 20 October 2023

1. Introduction

As the integration of hardware and software continues to evolve, production systems are becoming increasingly intricate, now often referred to as Cyber-Physical Production Systems (CPPS). In particular, Artificial Intelligence (AI) can be instrumental in improving processes such as anomaly detection, optimization, or predictive maintenance. However, at present, incorporating AI algorithms into these systems is far from straightforward; it demands substantial time, financial resources, and expertise. Adopting a standardized architecture could facilitate the integration of AI technologies, especially for small and medium-sized enterprises, empowering them to remain competitive.

For modern production processes, AI methods are used for various purposes. These include optimization, predictive maintenance, anomaly detection, and condition monitoring. The Cognitive Architecture for Artificial Intelligence (CAAI) was developed because the existing architectures from the domain of automation—i.e., the Reference Architecture Model Industrie 4.0 (RAMI4.0) [1], the Industrial Internet Reference Architecture (IIRA) [2], and the 5C architecture [3]—are too abstract. They do not define implementation details,

such as interfaces. Other cognitive architectures based on cognitive psychology, such as Soar [4] or ACT-R [5], lack generalizability. The KOARCH project endeavored to address the gap of crafting an architecture characterized by a low level of abstraction while maintaining the highest possible degree of generalizability, simultaneously.

The CAAI was introduced in [6] as a novel cognitive architecture for AI in CPPS. The goal of the architecture was to reduce the implementation effort for the usage of AI algorithms. The core of the CAAI is a cognitive module that processes the user's declarative goals, selects suitable models and algorithms, and creates a configuration for the execution of a processing pipeline on a Big Data Platform (BDP). Constant observation and evaluation against performance criteria assess the performance of pipelines for many different use cases. Based on these evaluations, the pipelines are automatically adapted if necessary. The modular design with well-defined interfaces enables the reusability and extensibility of pipeline components. A BDP implements this modular design, supported by technologies such as Docker, Kubernetes, and Kafka, for virtualization and orchestration of the individual components and their communication. The implementation of the architecture is evaluated using a real-world use case. The prototypic implementation is accessible on GitHub and contains a demonstration (https://github.com/janstrohschein/KOARCH, accessed on 16 October 2023).

During the retrospective evaluation of the CAAI, it became apparent that the architecture in the use case for CPPS provides a suitable environment for the implementation of Online Machine Learning (OML) algorithms. This is caused by the continuous data streams produced by the system's machines. In an OML set up, these enormous amounts of data would not need to be stored. Furthermore, the algorithms could be continuously updated and, thus, react better to structural changes within the data. For this reason, this article focuses on evaluating whether OML algorithms are a significant enhancement to the existing architecture. Furthermore, the possibilities of surrogate-model-based hyperparameter tuning in the context of OML algorithms will be explored.

This article is structured as follows: Section 2 begins with a summary and a brief retrospective evaluation of the CAAI. In addition, this section introduces the concept of OML and establishes a framework for performing experiments that compare Batch Machine Learning (BML) and OML. Following this, we detail the concept of Surrogate-Model-Based Optimization (SMBO), which we use to determine the best hyperparameters for the respective OML algorithms. Section 3 showcases the results of real-world application experiments conducted to evaluate the benefits of implementing OML in the CAAI. Subsequently, the outcomes of the Hyperparameter Tuning (HPT) are further described. A discussion and a conclusion to the article are provided in Sections 4 and 5.

2. Materials and Methods

2.1. The Cognitive Architecture CAAI

The CAAI was developed in the research project "Kognitive Architektur für Cyberphysische Produktionssysteme und Industrie 4.0" (KOARCH). KOARCH was established due to the lack of standardized architectures for AI applications in the industry. As a result, specialists often develop and implement their own architectures for various problems, which can be complex and costly. The KOARCH project aims to develop a reference architecture for production systems that can track a given use case (optimization, predictive maintenance, etc.), using an AI-based cognitive module. The architecture should meet general requirements, such as reliability, flexibility, generalizability, and adaptability. Furthermore, 12 specific requirements were defined at the beginning of the project [6]. These requirements are listed in Table 1.

The idea of the KOARCH project was to develop an architecture that meets these requirements, based on a BDP. This BDP is mapped in Figure 1. By looking at the structure of the architecture, it can be seen that the different components are realized via different modules. This modularization of the CAAI allows for flexible adaptation or extension. A virtualized container is used for every module or component of the BDP. Kubernetes is

used as a framework to orchestrate the different micro-services. The BDP is divided into two main layers: the conceptual layer and the data processing layer. The modules of these layers communicate via three different buses. Raw data from the CPPS, demonstrators, or external simulation modules enter the BDP via the data bus. Cleaned and preprocessed data are transported back to the data bus by the preprocessing module, to be forwarded to other modules of the data processing layer. The analytics bus ensures connection between the modules of the data processing layer and the conceptual layer. Results transmitted here have significantly higher information density than those from the data bus. The most influential component is the cognition module, located in the conceptual layer. It compiles pipelines for suitable algorithms based on overall goals and boundary conditions. The knowledge bus is responsible for communication between the Human–Machine Interface (HMI) and the conceptual layer, transmitting user commands and goals, as well as for reporting of the process and results to the user.

Table 1. Requirements, as specified by Bunte et al. [6].

Requirement	Description
R.1	The specified interfaces are well defined.
R.2	Strategies to select a suitable algorithm.
R.3	The system learns from experiences.
R.4	The software provides a thorough knowledge representation.
R.5	The system can acquire data from distributed systems.
R.6	The architecture stores and manages acquired process data and models.
R.7	The platform performs data preprocessing.
R.8	The system learns a model from data (might be time and resource-limited).
R.9	The platform performs a model analysis, which might have a limited response time.
R.10	The user is able to interact with the software.
R.11	The user is able to make decisions.
R.12	The user can apply actions on the control logic.

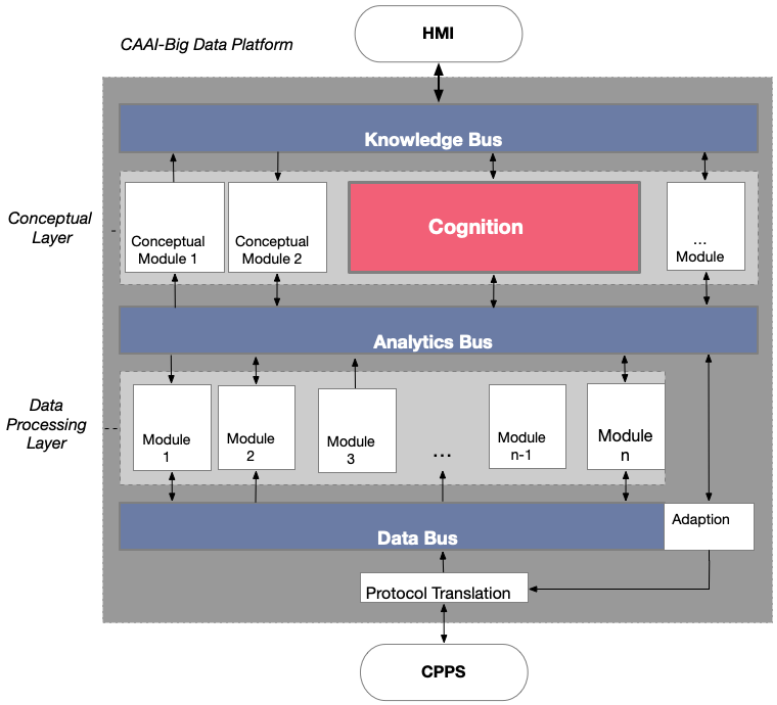

Figure 1. Structure of the BDP. Details can be found in Fischbach et al. [6].

The idea is that the users can define one or more higher-level goals, such as optimization, anomaly detection, predictive maintenance, etc. Additionally, they can establish constraints for the process. These constraints include the definition of signals that the algorithm utilizes as features for the respective algorithm, as well as limits for the associated values. Moreover, target-dependent settings can be configured. For instance, in an

optimization application, the objective function and the optimization goal (minimization or maximization) can be specified.

During the subsequent step, the cognitive module forms processing pipelines based on the selected objective. It selects appropriate preprocessing steps and algorithms. For decision making, the cognitive module relies on simulation data and experience from similar, previously developed applications. Moreover, SMBO has been implemented, to achieve maximum performance of the algorithms.

The evaluation of the architecture was performed using a Versatile Production System (VPS). This VPS is located in a laboratory of the Ostwestfalen-Lippe University of Applied Sciences (TH OWL). The VPS is a modular production system that processes corn, to produce popcorn and package it. The use case involves optimizing the efficiency of the popcorn packaging process. The goal is to reduce the amount of corn used and to maximize the volume of the popcorn. To achieve this objective, various algorithms, including Regression Trees, Kriging models, and other algorithms in BML configurations, were benchmarked. It is important to note that the VPS is solely a test setup within the smart factory at the university. It does not represent a real-world production system and is exclusively utilized for demonstration purposes. During the retrospective evaluation of the architecture, it became apparent that the CAAI performs well for fixed tasks. However, an individual implementation effort is still required for each use case. These findings will be detailed in a forthcoming paper.

During the detailed analysis of the architecture, it was observed that a critical category of algorithms, namely OML algorithms, had not been incorporated in the implementation. The configuration of the CPPS creates an ideal environment for the deployment of OML algorithms. The production system machinery generates continuous streams of data. Implementing OML strategies would mean these vast quantities of data would not need to be stored, alleviating storage demands. Furthermore, the system could flexibly and swiftly adapt to concept drifts. Therefore, the primary focus of this article was to evaluate the potential and effectiveness of integrating OML algorithms within the CPPS context and to demonstrate the importance of integrating OML capabilities into the CAAI framework. The comparison of BML and OML in the context of a real-world application is unprecedented in this field of research.

2.2. The Need for Online Machine Learning

The volume of data generated from various sources has increased enormously in recent years (“Big Data”). Technological advances have enabled the continuous collection of data. Sensor data, web data, social media data, share prices, search queries, clickstream data, operational monitoring data, online advertising, mobile data, and the Internet of Things data are referred to as streaming data. Streaming data, or streams of data, is an infinite and continuous flow of data from a source, often arriving at very high speeds. Therefore, streaming data is a subset of Big Data. In addition to the increased frequency, streaming data and static data also differ, in that the former have less structure. Streaming data are loosely structured, volatile (only available once), and always “flowing” data. They require real-time or near-real-time analysis. As the data stream is constantly being produced and never ends, it is not possible to store this enormous amount of data and only then carry out analyses on it (as with batch data).

The classical BML approach boils down to the following [7]:

1. Loading and pre-processing the train data;
2. Fitting a model to the data;
3. Calculating the performance of the model on the test data.

This procedure has certain disadvantages. Batch learning models are not suitable for handling streaming data, since multiple passes over the data are not possible. The batch models may soon become outdated due to concept drifts (i.e., data distribution changes over time). Furthermore, BML has problems regarding storage requirements, unknown data, and accessibility of data, which will be discussed next.

For example, in the case of energy consumption forecasts, the previously known consumption values are only one element that is required for the modeling. In practice, future demand is driven by a range of non-stationary forces—such as climate variability, population growth, or disruptive clean energy technologies—that may require both gradual and sudden domain adjustment. Therefore, prediction, classification, regression, or anomaly detection approaches should be able to detect and respond to conceptual deviations in a timely manner, so that the model can be updated as quickly as possible. Although BML models can be retrained regularly, this is infeasible in many situations because the training is too expensive.

Another problem for BML is that it cannot incorporate new data containing unknown attributes. When new data are made available, the model has to be learned from scratch with a new dataset composed of the old data and the new data. This is particularly difficult in a situation where new data and attributes come in every day, every hour, every minute or even with every measurement, as is the case for production processes.

The enormous amount of data can lead to another problem, where the dataset size exceeds the available amount of RAM. Possible solutions include the optimization of data types (sparse representations), the usage of a partial dataset ("out-of-core learning"), i.e., the data are divided into blocks or mini-batches, or the application of highly simplified models.

Last but not least, data accessibility is a problem for BML: each time the BML model is trained, features must be extracted. The problem is that some features are no longer available after some time, e.g., because they were overwritten or simply deleted. This means that features that were still available last week may no longer be available at the current time. In general, it is not always possible to provide all data at the same time and in the same place. In addition to these issues, BML and especially deep learning algorithms can cause high energy costs.

The challenges of streaming data led to the development of a class of methods known as incremental or online learning methods. The introduction of different methods of online learning/incremental learning has been quite slow over the years, but the situation is now changing [8–10]. The point of incremental learning is to fit an ML model to a data stream. In other words, the data are not available in their entirety, but the observations are provided one at a time. This way, the models can be updated incrementally before the data are discarded. The axioms for stream learning, which form the foundation of OML, can be derived from the following requirements [9]:

1. Each instance can only be used once;
2. The processing time is severely limited;
3. The memory is limited ("sublinear in the length of the stream");
4. The algorithm must be able to deliver a result at any time ("anytime property");
5. Data streams are assumed to change over time, i.e., the data sources are not stationary.

2.2.1. OML Methods

There are many ML models that can be adapted to OML implementations. For example, online linear regression is a popular method in OML. Stochastic Gradient Descent (SGD) is used to update the coefficients in the implementation of an online linear regression model, as not all data are available at once. SGD is commonly used to train neural networks.

Tree-based algorithms are also popular in OML. Trees have nodes for testing attributes, usually by comparison, and branches for storing the test results and making predictions (of a class in classification or a value in regression). One challenge with streaming data is the high storage requirements, as it is impossible to save all data. Trees allow for compact representation, making them popular methods in OML. A BML tree reuses instances to calculate the best splitting attributes ("splits"). Therefore, using BML decision tree methods like Classification And Regression Tree (CART) is not effective in a streaming data context. Instead, Hoeffding trees are used in OML. They do not reuse instances but wait for new instances to arrive [11]. As an incremental decision tree learner, a Hoeffding tree is better suited to a streaming data context. It is based on the idea that a small sample is often

sufficient to select an optimal splitting attribute, supported by the statistical result known as the Hoeffding bound. The Hoeffding tree converges to a tree generated by a BML algorithm with sufficiently large data [9]. However, streaming data can be very noisy, affecting performance (in terms of prediction accuracy) and potentially generating very large trees. Several extensions of the basic Hoeffding tree exist, such as Hoeffding Anytime Trees that work similarly but use a modified splitting procedure.

The Hoeffding Adaptive Tree (HAT) [12] is an extension of the Hoeffding tree, incorporating a mechanism for identifying concept drift. It employs an instance of the ADWIN [13] concept drift detector at each decision node, to monitor potential shifts in data distribution.

2.2.2. The Evaluation Frame: How to Compare the Methods

To compare OML to classical approaches, and to evaluate the strengths and weaknesses of various forecasting methods, different approaches were employed in our experiments. Three distinct methods were used to generate the training dataset for batch procedures. In order to provide a detailed explanation of these methods, it is important to first describe the experimental procedure. The objective of each method was to produce point forecasts with maximum accuracy for a predetermined horizon of 150 data points in the future. However, the development of models differed among the methods. For batch procedures, a classical train–test split was used, referred to as "batch" hereafter, along with a train–test split using a landmark approach and a train–test split with a shifting window. Additionally, the OML approach was utilized for model development. The evaluation functions `eval_bml`, `eval_bml_landmark`, `eval_bml_window`, and `eval_oml_horizon` accept two data frames as arguments:

- train, denoted as D_{train}, with size s_{train}, is used to fit the model;
- test, denoted as D_{test}, with size s_{test}, is used to evaluate the model on new (unseen) data.

First, the method `eval_bml` implements the "classical" BML approach. The algorithm is trained once on the training dataset, resulting in a model, denoted as M_{bml}, that is not modified. The model M_{bml} is evaluated on the test data, where the horizon, denoted as $h \in [1, s_{\text{test}}]$, specifies the size of the partitions that D_{test} is split into. If $h = s_{\text{test}}$, then the basic ML train–test setting is implemented. If $h = 1$, an OML setting is simulated.

Second, the method `eval_bml_landmark` implements a landmark approach. The first step is similar to that of the BML approach, resulting in an initial model $M^{(1)}_{\text{bml}}$. However, subsequent steps differ: after making a prediction with $M^{(1)}_{\text{bml}}$ for the batch of data instances from the interval $[s_{\text{train}}, s_{\text{train}} + h]$, the algorithm is retrained on the interval $[1, s_{\text{train}} + h]$, to produce an updated model $M^{(2)}_{\text{bml}}$. During the third step of the landmark BML approach, $M^{(2)}_{\text{bml}}$ makes predictions for $[s_{\text{train}} + h, \text{train} + 2 \times h]$, and a new algorithm, $M^{(2)}_{\text{bml}}$, is trained on $[1, \text{train} + 2 \times h]$.

Third, the method `eval_bml_window` implements a window approach. Again, the first step is similar to that of the BML approach, resulting in an initial model, $M^{(1)}_{\text{bml}}$. Subsequent steps are similar to those of the landmark approach, with one important exception: instead of being trained on the complete set of seen data, the algorithm is trained on a moving window of size s_{train}.

Finally, the method `eval_oml_horizon` implements an OML approach. This approach differs fundamentally from BML approaches because every single instance is used for both prediction and training. If $h = 1$, a "pure" OML algorithm is implemented. If $h > 1$, OML computations are performed h times.

A summary of the training and test set generation process, related to the corresponding evaluation procedure, can be found in Table A1 in the Appendix A. Additionally, Figure 2 visualizes the differences between the evaluation techniques introduced. Several criteria were used to evaluate the different approaches, including Mean Absolute Error (MAE), computation time, and memory consumption. The selection of these metrics was based on the different requirements that an end user might have for the system. While the model with the lowest error was preferred, computation time can be a crucial factor in high-frequency

data, and memory consumption should not be ignored, as more complex models can take up several gigabytes. By memory consumption, we do not mean an exact calculation of the size of the model, but measurements of peak memory consumption during the model's training and testing processes. This approach allowed us to conveniently compare the memory consumption of ML algorithms from different Python packages (*sklearn* and *River*). All the evaluation methods described in this section are available in the open-source *spotRiver* package on GitHub (https://github.com/sequential-parameter-optimization, accessed on 16 October 2023).

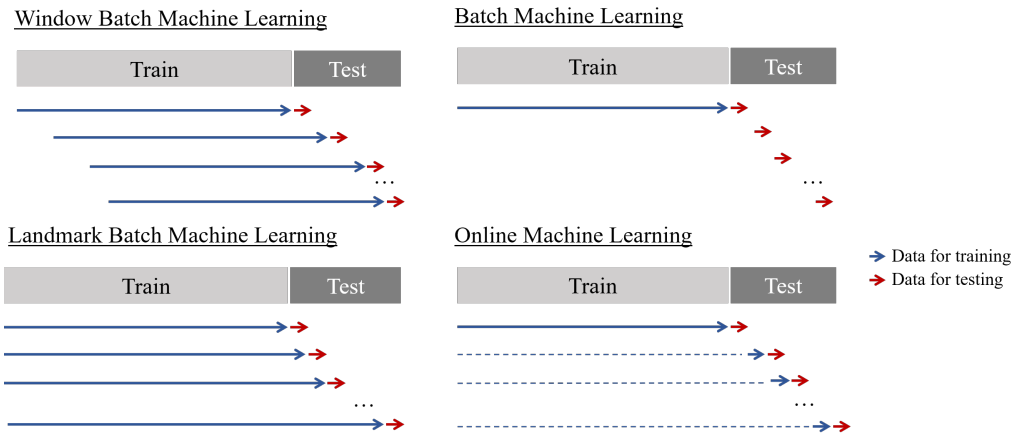

Figure 2. Construction of the different training datasets, depending on the modeling approach chosen.

2.2.3. Real-World Application: Slitting Machines

In the experiments discussed in this work, data were collected using a test setup for winding stations from "Kampf Schneid- und Wickeltechnik GmbH & Co. KG", Wiehl, Germany, a company that specializes in building machines for slitting and winding web-shaped materials such as paper, plastic films, or aluminum foil, as depicted in Figure 3. A paper core is secured between two winding stations, to wind the web into a roll, achieving a diameter of up to 1500 mm and weights of up to 6 tons. The necessary web tension for different materials is maintained by a drive, which adjusts to compensate for the increasing diameter.

The test setup facilitated the evaluation of new concepts for winding stations and helped in determining the lifespan of various components, including bearings and belts. Additional sensors were installed to monitor temperatures and vibration levels at different points, enabling a more comprehensive analysis of their behavior. In one of the trials, a machine run to wind up a representative material was simulated under defined conditions, including parameters such as material thickness, material density, material width, induced tension, machine speed, and acceleration and deceleration times, in addition to core and finished diameters.

For the current experimental setup, sensor data from the winding machine were collected locally and supplied by the company. Machine learning algorithms were executed on a separate device. In the future, the plan is to establish a bi-directional connection between the sensors and the computing device.

The collected time-series data encapsulated information regarding motor temperature, revolutions, and torques, as well as data from external sensors monitoring temperature and vibration levels. For the experiments outlined in this article, only specific data were utilized: motor revolution $[\frac{1}{\text{min}}]$, motor torque [Nm], and the vibration data at a particular point. The vibration data were measured in analog values between 0 and 27,648, related to a range between 0–25 $[\frac{\text{mm}}{\text{s}^2}]$. Motor revolution and torque were used as input features, while the vibration level was the prediction target. Data were logged every one hundredth of a second, with each timestamp recording the respective feature values. To simulate external influences, such as roll handling or other factory operations, further vibrations were introduced intermittently by gently hitting and shaking the winding stations.

Figure 3. Visualization of the slitting machine that was used to generate the data for the experiments. The image is provided by 'Kampf Schneid- und Wickeltechnik GmbH & Co. KG', Wiehl, Germany.

In our experiments, the data provided by Kampf Schneid-und Wickeltechnik GmbH & Co. KG were utilized, to predict the vibration of level station 1, using motor revolution and torque. A forecast horizon of t = 150 was defined. This horizon refers to the time when the models will be updated again. For instance, in this experiment, the models were updated after collecting 150 data points, although each approach employed a distinct strategy. The classic batch method did not update the model and only utilized training data. The batch approach with a landmark strategy included newly collected data in the training data and created a new model on the enlarged dataset. On the other hand, the batch model with a shifting window approach excluded the first 150 data points of the training set and appended each new 150 data points to the end of the training data. A new model was then produced based on this new dataset, ensuring that the length of the training dataset remained constant. Finally, in the OML approach, 150 data points were collected and sequentially passed to the model.

In the batch approaches, a Decision Tree Regressor from the *sklearn* package [14] was utilized, while the Hoeffding Tree Regressor (HTR) from the *River* package [8] was used in the OML approach. Before passing data to the regressor, it was standardized. For the initial training of the models, 1,462,555 samples were used. The subsequent evaluation horizon consisted of 30,196 data points. This test set included four potential future evolutions of the vibration level. Figure 4 displays the temporal evolution of the vibration observed in one test scenario, revealing distinct local peaks followed by sharp drops and subsequent slow recovery before the emergence of new peaks.

2.3. Hyperparameter Tuning

The goal of HPT is to optimize the hyperparameters in a way that improves the performance of an ML model. This is an important but usually difficult and computationally intensive task. The simplest approach—but also the most computationally expensive—is a manual search (or trial and error) [15].

Common approaches include the Simple Random Search (RS), where hyperparameters are randomly and repeatedly selected for evaluation, and the grid search. Directed search methods and other model-free algorithms, such as evolution strategies [16] or pattern search [17], also play an important role. Hyperband, a multi-armed bandit strategy that dynamically allocates resources to a set of random configurations and uses successive bisections to stop configurations with poor performance [18], is also commonly used in the HPT domain. The most sophisticated and efficient approaches are Bayesian Optimization (BO) and SMBO methods, which are based on the optimization of cost functions obtained through simulations or experiments.

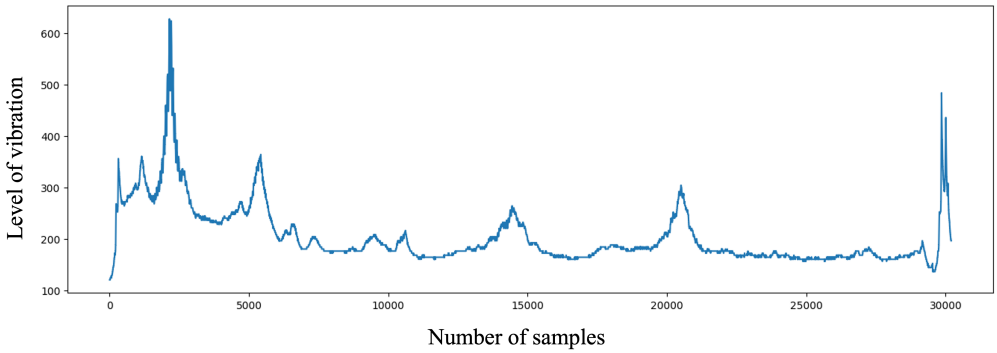

Figure 4. Recorded vibration for each data sample in the test set. The X-axis represents the number of the current sample in the test set, whereas the Y-axis is related to the vibration. The vibration data were measured in analog values between 0 and 27,648, related to a range between 0–25 [$\frac{mm}{s^2}$]. The peaks and drops in the time series were intentionally induced by shaking and striking the machine during the measurement process.

For this article, we considered an HPT approach based on the Sequential Parameter Optimization Toolbox (SPOT) [7], which is suitable for situations where only limited resources are available. This may be due to limited availability or cost of hardware. Another reason might be that confidential data may only be processed locally, due to legal requirements. Furthermore, our approach emphasizes the importance of understanding algorithms as a key tool for transparency and explainability. This can be enabled by quantifying the contribution of ML and Deep Learning components (nodes, layers, split decisions, activation functions, etc.) and understanding the meaning of hyperparameters and their interactions. The SPOT provides statistical tools for understanding hyperparameters and their interactions. Additionally, the SPOT software code is available in the open-source *spotPython* and *spotRiver* packages on GitHub (https://github.com/sequential-parameter-optimization, accessed on 16 October 2023), allowing for replicability of results. The SPOT is established open-source software that has been maintained for over 15 years [7]. It includes SMBO methods for tuning, based on classical regression and analysis of variance techniques, tree-based models such as CART and RF, BO (Gaussian Process Models, also known as Kriging), and combinations of different meta-modeling approaches. Any ML model in scikit-learn (*sklearn*) can be used as a meta-model.

The loop of the model-based tuning process with the SPOT can be divided into the following steps [7]:

- Setup: Different combinations of hyperparameter values are evaluated, in order to build an initial design.
- Evaluation: The new or initial hyperparameters are evaluated.
- Termination: The loop checks weather a termination criterion like maximum number of iteration or maximum tuning time has been reached.
- Selection: This step selects samples for building the surrogate model.
- Building Surrogate: The surrogate is built.
- Surrogate Search: The algorithm searches for the best hyperparameter settings based on the surrogate model.
- Optimal Computing Budget Allocation (OCBA): This step is used to determine the number of repeated evaluations.

2.3.1. SMBO Tuning Setup

In order to find the optimal parameters for the HRT algorithm, we employed Gaussian Process Regression (GPR) [19] as a surrogate model in the context of SMBO. GPR, also known as the Kriging model, is the default surrogate choice for SMBO within the SPOT framework. To drive this optimization process, we chose the Differential Evolution (DE) [20] algorithm as our optimization method.

The optimization bounds, along with the default values assigned to the tuned hyperparameters, are outlined in Table 2. Notably, the optimization process is time limited, with a maximum duration of 100 min. It is important to emphasize that this countdown starts only after the initialization of the initial surrogate model design, which in our specific case consisted of 50 data points. Due to the fact that our dataset was very large (almost 1.5 million samples), we used only 2% of the trainings and test set for the HPT. The remaining data were subsequently used for the actual training and evaluation of the tuned and default model. The goal of the optimization is to minimize a combined value of MAE, computation time, and memory usage. These values are weighted differently (MAE: 1, memory consumption: 1×10^{-3}, calculation time: 1×10^{-3}).

Table 2. A list of the hyperparameters considered for the tuning of the Hoeffding Tree Regressor. The default values, along with their respective lower and upper bounds, are provided, accompanied by a short description of each hyperparameter.

Parameter	Default	Lower Bound	Upper Bound	Description
grace_period	200	10	1000	Number of samples a leaf observes between the splits.
max_depth	20	2	20	Maximum depth of the tree.
delta	1×10^{-7}	1×10^{-10}	1×10^{-6}	The significance level for the Hoeffding bounds is calculated by $1 - \delta$.
tau	0.05	0.01	0.1	Threshold below which a split will be forced to break ties.
leaf_prediction	0	0	2	The prediction mechanism at the leaves (0—mean, 1—model, 2—adaptive).
leaf_model	0	0	2	The regression model that is used (0—linear regression, 1—PA-regressor, 2—perceptron).
model_selector_decay	0.95	0.9	0.99	The exponential decaying factor applied to the learning models' squared errors if the leaf prediction mechanism is 'adaptive'.
splitter	0	0	2	The splitter for defining thresholds and performing splits (0—EBSTS, 1—TEBSTS, 2—QO).
min_samples_split	5	2	10	Minimum number of samples a branch must have, resulting from a split.
binary_split	0	0	1	If True, only binary splits are allowed.
max_size	500	100	1000	Maximum size of the tree in MB.
memory_estimate_period	1×10^6	1×10^5	1×10^6	Number of instances between memory checks.
stop_mem_management	0	0	1	If true, stop growing the tree if maximum size (max_size) is reached.
remove_poor_attrs	0	0	1	If true, disable poor attributes.
merrit_preprune	0	0	1	If true, enable merit-based pre-pruning.

3. Results

3.1. Real-World Application Results

This chapter presents the results of the different evaluation procedures that were presented in Section 2.2.2. In Figure 5, we see a comparison between model predictions and actual values. To look at the behavior of the methods in more detail, the graph refers only to a window of 3000 data points. All three BML methods consistently overshot the actual values, while the OML algorithm generally provided accurate predictions. The first 500 predictions showed clear oscillations of the actual values. Especially with these data points, batch learning evaluation methods have difficulty making usable predictions. It is clear to see how the OML algorithm responded flexibly to the fluctuations and delivered accurate predictions.

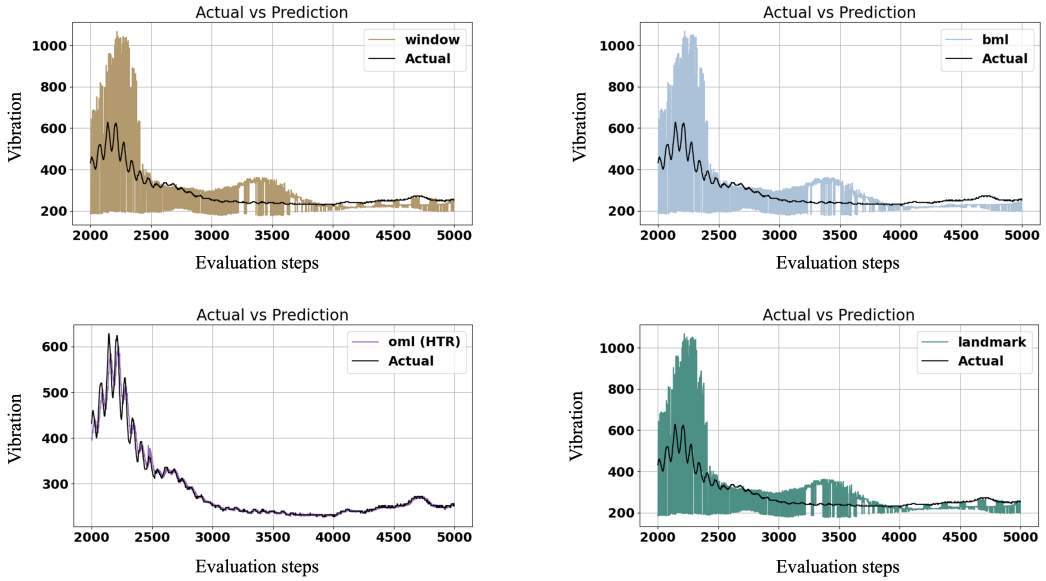

Figure 5. The comparison between predicted and real values for a subset of the evaluation horizon. Each subplot is related to the predictions of one of the evaluation strategies. The Y-axis shows the vibration values, whereas the X-axis shows the evaluation steps. Please consider the different scales of the Y-axes.

3.1.1. Evaluation Metric

The performance of the different approaches is visualized in the top graph of Figure 6. It shows how the MAE evolved over the evaluation horizon. All the batch learning evaluation methods produced comparable results. Initially, the performance degraded slightly and then improved continuously. The OML approach comparatively achieved constant results and outperformed the batch evaluations over the entire horizon.

3.1.2. Computation Time

The second diagram in Figure 6 shows a comparison of the computation times of the different methods. As assumed, the landmark and shifting-window methods showed a continuous increase in computation time, due to the models' need to be retrained at each evaluation iteration. By contrast, the conventional batch learning approach exhibited a much lower processing time because of its singular model training phase. On the other hand, the OML algorithm achieved time-efficient results as well. This was because OML updates models incrementally, rather than training from scratch with each evaluation.

3.1.3. Memory Consumption

The lowest graph of Figure 6 shows the memory consumption. Here, the OML approach also delivered results comparable to the basic batch approach. However, it should be emphasized again that the batch approach's memory consumption only took place during the training step, and the remaining consumption was negligible. This fact is also visualized by the graph. In the first evaluation step, the memory consumption for the classic batch method dropped towards zero. The shifting window and the landmark approach performed comparably poorly. This was mainly due to the generated model, which had to be built again in each iteration.

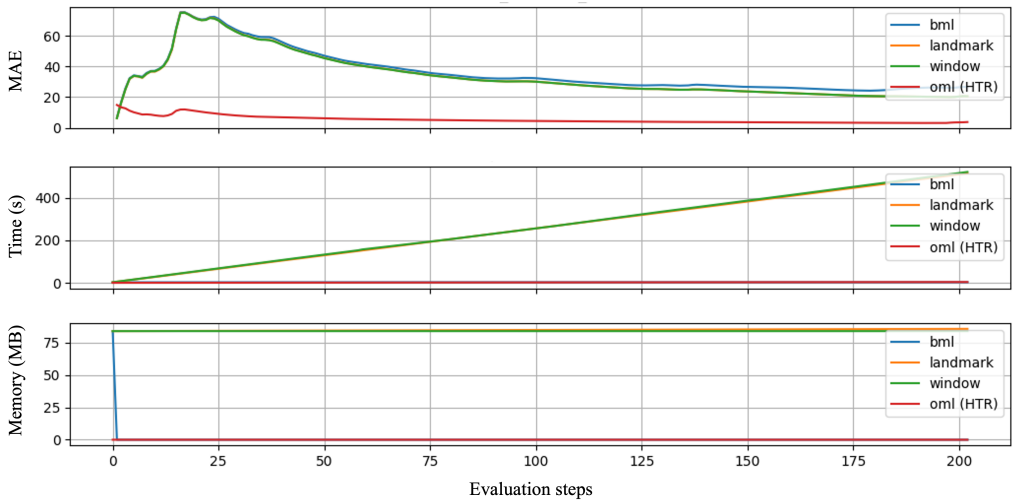

Figure 6. The MAE, computation time, and memory consumption of different approaches for each evaluation step. The MAE plot shows an overlap of the graphs of all BML methods. In the plots for computation time and memory consumption, the curves of landmark and shifting window, as well as the OML and the classical batch method overlap.

3.1.4. Statistical Evaluation

Given that graphical evaluations alone may not be sufficient, a statistical evaluation was conducted, to determine whether there was a significant difference in the mean deviation of the differences between the prediction and the true values of two time series. This approach aimed to ascertain whether one of the time series provided a better approximation of the true time series by examining if the mean difference of one time series was significantly smaller than the other. To perform the statistical analysis, the procedure of [21] was used.

Our process involved selecting two time series for comparison, where time series one was always the OML approach and time series two was one of the remaining methods. Next, the number of samples required to make an informed decision was calculated, based on the following assumptions:

- $\alpha = 0.05$—probability of a Type I error, known as a false positive;
- $\beta = 0.2$—probability of a Type II error, known as a false negative; $1 - \beta$ is known as the power of the test;
- $\sigma =$ standard deviation of the difference of the absolute differences between model 1 and model 2;
- $\Delta =$ twice the mean value of the absolute deviation between the actual values and the output of the OML approach.

Instead of considering each sample value individually, we considered the overall mean of the absolute deviation between the prediction and the true value from each time series. It was determined that between 518 and 527 samples were needed, depending on the time series. As the analyzed time series comprised around 30,000 samples, obtaining the required sample size was not problematic.

In the third step, a one-sided *t*-test was conducted, using the difference of the absolute values between the two time series. We formulated the following hypothesis, where j denoted the index of either the batch, landmark, or shifting window approach:

$$H_0 : \mu_{OML} - \mu_j \geq 0$$
$$H_1 : \mu_{OML} - \mu_j < 0.$$

Based on the analysis, it was concluded that the null hypothesis could be rejected in all cases, indicating that the mean deviation of the differences of the OML approach was significantly smaller than that of all the other approaches.

3.2. Hyperparameter Tuning Results

This section highlights the results of the HPT process. Figure 7 shows the progress of the tuning process. The initial design phase is represented by the 50 black data points on the left side of the graph. Each point represents the result of the evaluation at different stages of our initial design phase. The more data points that are evaluated for the initial design, the more accurate our surrogate model will be at the start of the optimization process. The continuous red line that follows the initial points traces the trajectory of the best hyperparameter setup found so far within the tuning progresses. It can be seen that the value of the objective function could be reduced after initialization through optimization.

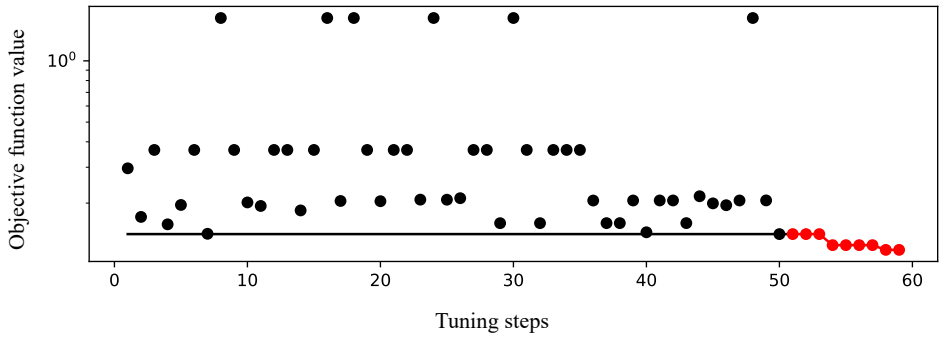

Figure 7. Visualization of the hyperparameter tuning progress. The Y-axis represents the values of the objective function, whereas the X-axis relates to the tuning steps. The solutions from the initial design are black, and the subsequent solutions from the optimization process are red.

Figure 8 illustrates the comparison of results over the entire evaluation horizon, in terms of MAE, computation time, and memory consumption between the tuned model and the model with default parameters. It is evident that the tuned model consistently outperformed the default algorithm, in terms of MAE, throughout the evaluation period. However, it is noteworthy that the tuned model consumed more computation time and memory resources than the default model. This can be explained by the fact that, as explained in Section 2.3.1, the weighting of the MAE in this particular tuning run was significantly higher than the weighting of the time and memory consumption.

To statistically prove the improvement in performance, with respect to the metric, a one-sided *t*-test, as explained in Section 3.1.4, was performed again, with the following hypothesis:

$$H_0 : \mu_{HTR_{tuned}} - \mu_{HTR_{default}} \geq 0$$
$$H_1 : \mu_{HTR_{tuned}} - \mu_{HTR_{default}} < 0.$$

The result of the *t*-test stated that the zero hypothesis had been rejected. This meant that the average deviations of the predictions of the tuned model were significantly lower than those of the default model.

The increase in computation time and memory consumption can be attributed to the values of the tuned parameters. Table 3 presents a comparison between the tuned and the default values for each hyperparameter, along with their respective importance, as determined during the SPOT optimization. It is evident that the leaf model and the prediction function of the leaves exerted the most significant influence on the evaluation outcomes. Upon contrasting the tuned and default values, the reasons behind the higher memory consumption and computation time in the tuned model became apparent. Specifically, the grace period, set at a value of 12, notably deviated from the default model's value of 200. Consequently, a considerably greater number of splits occurred in the tuned model.

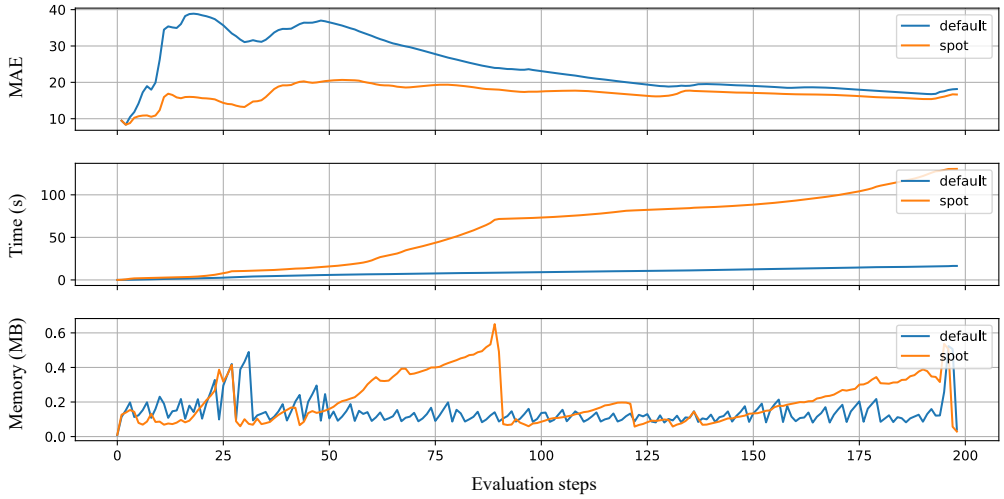

Figure 8. Evaluation results of the tuned and untuned HTR models for mean absolute error, computation time, and memory consumption. The X-axes are related to the evaluation steps. The tuning was performed with a focus on improving the performance (MAE).

Table 3. Comparison of default and tuned hyperparameters for improved performance of the HTR. The column on the far right illustrates the determined importance of the individual parameters during the tuning process.

Parameter	Default	Tuned	Importance
grace_period	200	12	0
max_depth	20	20	0.00
delta	1×10^{-7}	1×10^{-6}	0.00
tau	0.05	0.01	0.00
leaf_prediction	mean	adaptive	88.66
leaf_model	linear regression	linear regression	100.00
model_selector_decay	0.95	0.99	0.00
splitter	EBSTS-Splitter	EBSTS-Splitter	0.17
min_samples_split	5	8	0.00
binary_split	0	1	0.00
max_size	500	987.96	0.00
memory_estimate_period	1×10^{6}	2.6×10^{5}	0.00
stop_mem_management	0	0	0.00
remove_poor_attrs	0	0	0.00
merrit_preprune	0	0	0.00

Hyperparameter Tuning for Improving Time and Memory Consumption

In production processes, certain conditions may necessitate the optimization of algorithms, with respect to time and memory consumption. These conditions may include a requirement for near-real-time capability, limited RAM capacities, or the need to conserve energy. Consequently, we conducted an overhead experiment, to demonstrate that the HTR can be optimized for these two criteria, using SPOT. We adjusted the weightings as described in Section 2.3.1 to prioritize optimization (MAE: 1, memory consumption: 10, calculation time: 10).

Figure 9 illustrates that, over the entire evaluation horizon, the computation time and memory consumption of the tuned model were significantly lower than those of the default model. However, it is also evident that the tuned model exhibited a higher deviation in the predictions of vibration development due to the altered optimization priority.

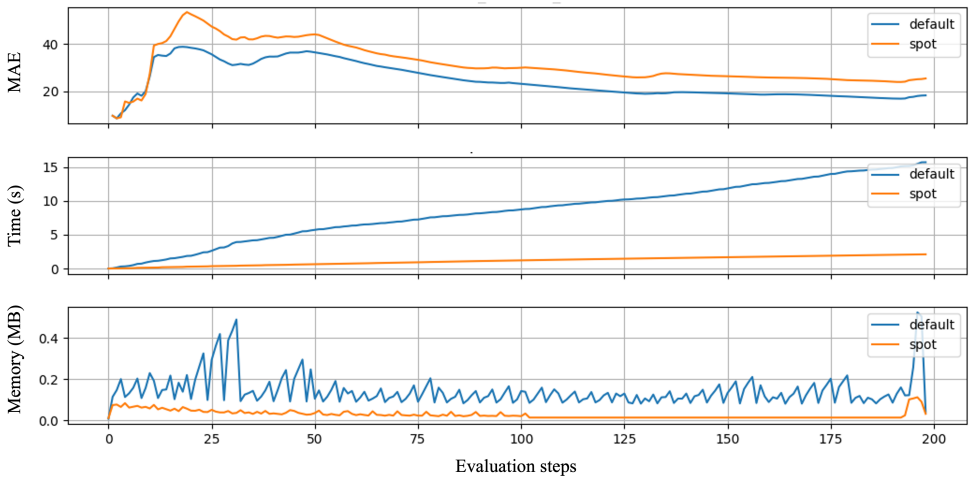

Figure 9. Evaluation results of tuned and untuned HTR models for MAE, computation time, and memory consumption. The X-axes are related to the evaluation steps. The tuning was performed with a focus on reducing time and memory consumption.

This behavior can be elucidated by examining Table 4. It is apparent that the grace period is nearing the upper boundary established in the tuning process. Consequently, the splitting process of the leafs was not executed as frequently as for the untuned model, thereby conserving computational resources. Moreover, the maximum depth of the tree at a value of four was markedly lesser than the default setting of the algorithm, which stood at 20. This further contributed to the reduction in computational effort and memory usage.

Table 4. Comparison of the default and tuned hyperparameters of the HTR for improved calculation time and memory consumption. The column on the far right illustrates the determined importance of the individual parameters during the tuning process.

Parameter	Default	Tuned	Importance
grace_period	200	937	0
max_depth	20	4	0.00
delta	1×10^{-7}	1×10^{-6}	0.00
tau	0.05	0.1	0.00
leaf_prediction	mean	adaptive	88.66
leaf_model	linear regression	linear regression	100.00
model_selector_decay	0.95	0.9	0.00
splitter	EBSTS-Splitter	EBSTS-Splitter	0.17
min_samples_split	5	2	0.00
binary_split	0	1	0.00
max_size	500	854.73	0.00
memory_estimate_period	1×10^{6}	1.3×10^{6}	0.00
stop_mem_management	0	0	0.00
remove_poor_attrs	0	0	0.00
merrit_preprune	0	0	0.00

4. Discussion

In this study, we revisited the CAAI developed in the KOARCH project. During the revision of this architecture, we briefly noted the existing limitations to automating the algorithm pipeline development process for all environments and use cases. This was mainly due to rapidly changing software interfaces and transfer complexities. While this highlighted the need for some human intervention, our focus quickly shifted to exploring the potential of OML in the context of production processes. Our attention centered on the notable absence of OML technology in the current CAAI framework, especially given the continuous data streams in CPPS. To illustrate the benefits of OML in industrial setups,

we conducted experiments using a slitting machine setup. During testing, we compared the OML approach to three BML-based methods. The OML approach significantly outperformed the BML strategies, primarily due to its ability to update models iteratively, allowing for a swift response to concept drift in data. This feature is especially useful in machinery and predictive maintenance applications, presenting a more efficient alternative. Furthermore, the OML approach demonstrated computational benefits, showing not only time efficiency but also conservative memory usage. The performance was notably better than that of the other two methods, where the models were retrained following the initial training—namely, the shifting window and landmark strategies. In order to optimize the OML algorithm's performance, we introduced SMBO, to identify the best hyperparameter sets. Using SPOT for hyperparameter tuning significantly improved algorithm performance. Additionally, adjusting the priorities during the optimization phase successfully optimized computation time and memory consumption. This study underscores the potential of integrating OML technology and the effectiveness of hyperparameter tuning in enhancing CPPS. This research paves the way for the development of more agile and efficient Cyber-Physical Production Systems, ready to tackle the dynamic demands of industrial operations with refined precision and adaptability.

5. Conclusions

In conclusion, this work can be summarized in three key findings:

First, the realization of a universal cognitive architecture is challenging to implement, particularly in the context of diverse interacting systems. This challenge arises from the absence of standardized interfaces and the continuously evolving nature of software components, necessitating human interaction.

Second, our results demonstrate that OML algorithms perform better than related BML methods, in terms of MAE. Additionally, online applications exhibit significant advantages, in terms of computational efficiency and memory consumption.

Third, this study underscores the crucial role of OML as an indispensable extension of existing CAAI architectures, particularly in light of the vast amounts of data generated by modern production processes.

Author Contributions: This paper was supervised by A.H., R.S. and T.B.-B. These three authors contributed equally to all the tasks. They were assisted in the following subtasks by the following authors: validation of the approach was performed by L.H., P.P. and C.G.; software was developed by N.P., N.B., A.S. and L.H.; data curation was performed by A.S. and N.B.; writing was supported by L.H. and M.R. All authors have read and agreed to the published version of the manuscript.

Funding: This research work was funded by the German Federal Ministry for Economic Affairs and Climate Action, as part of the project "*IMProvT_II*—Intelligente Messverfahren zur energetischen Prozessoptimierung von Trinkwasserbereitstellung und -verteilung II", funding code 03EN2086A.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available, due to confidentiality restrictions.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

BDP	Big Data Platform
BML	Batch Machine Learning
BO	Bayesian Optimization
CAAI	Cognitive Architecture for Artificial Intelligence
CART	Classification and Regression Tree

CPPS	Cyber-Physical Production System
DE	Differential Evolution
EFDT	Extremely Fast Decision Trees
GPR	Gaussian Process Regression
HAT	Hoeffding Adaptive Tree
HMI	Human Machine Interface
HPT	Hyperparameter Tuning
HTR	Hoeffding Tree Regressor
OML	Online Machine Learning
RF	Random Forest
SGD	Stochastic Gradient Descent
SMBO	Surrogate-Model-Based Optimization
SPOT	Sequential Parameter Optimization Toolbox
VPS	Versatile Production System

Appendix A

Table A1. Evaluation methods. Batches are denoted as intervals, e.g., $[a, b]$. The OML approach passes every instance from the interval to the online algorithm separately for prediction and update (training).

Name	Step	Training Interval/Instances	Training Batch Size	Model	Prediction Interval
BML ("Classical")	1	$[1, s_{\text{train}}]$	s_{train}	$M^{(1)}$	$[s_{\text{train}}+1, s_{\text{train}}+h]$
	n	$[1, s_{\text{train}}]$	0	$M^{(1)}$	$[s_{\text{train}}+(n-1)\times h+1, s_{\text{train}}+n\times h]$
Landmark BML	1	$[1, s_{\text{train}}]$	s_{train}	$M^{(1)}$	$[s_{\text{train}}+1, s_{\text{train}}+h]$
	n	$[1, s_{\text{train}}+(n-1)\times h]$	$s_{\text{train}}+(n-1)\times h$	$M^{(n)}$	$[s_{\text{train}}+(n-1)\times h+1, s_{\text{train}}+n\times h]$
Window BML	1	$[1, s_{\text{train}}]$	s_{train}	$M^{(1)}$	$[s_{\text{train}}+1, s_{\text{train}}+h]$
	n	$[1+(n-1)\times h, s_{\text{train}}+(n-1)\times h]$	s_{train}	$M^{(n)}$	$[s_{\text{train}}+(n-1)\times h+1, s_{\text{train}}+n\times h]$
OML	1	$[1, s_{\text{train}}]$	1	$M^{(1)}$	$[s_{\text{train}}+1, s_{\text{train}}+h]$
	n	$[1, s_{\text{train}}+(n-1)\times h]$	1	$M^{(n)}$	$[s_{\text{train}}+(n-1)\times h+1, s_{\text{train}}+n\times h]$

References

1. Adolphs, P.; Bedenbender, H.; Dirzus, D.; Ehlich, M.; Epple, U.; Hankel, M.; Heidel, R.; Hoffmeister, M.; Huhle, H.; Kaercher, B. *Reference Architecture Model Industrie 4.0 (RAMI4.0)*; Tech. Rep.; VDI: Düsseldorf, Germany, 2015.
2. Lin, S.W.; Miller, B.; Durand, J.; Bleakley, G.; Ghigani, A.; Martin, R.; Murphy, B.; Crawford, M. *The Industrial Internet of Things Volume G1: Reference Architecture v1.80*; Technical Report; Industrial Internet Consortium: Boston, MA, USA, 2017.
3. Lee, J.; Jin, C.; Bagheri, B. Cyber physical systems for predictive production systems. *Prod. Eng.* **2017**, *11*, 155–165. [CrossRef]
4. Laird, J.E.; Newell, A.; Rosenbloom, P.S. SOAR: An Architecture for General Intelligence. *Artif. Intell.* **1987**, *33*, 1–64. [CrossRef]
5. Anderson, J.R. A Simple Theory of Complex Cognition. *Am. Psychol.* **1996**, *51*, 355–365. [CrossRef]
6. Fischbach, A.; Strohschein, J.; Bunte, A.; Stork, J.; Faeskorn-Woyke, H.; Moriz, N.; Bartz-Beielstein, T. CAAI—A cognitive architecture to introduce artificial intelligence in cyber-physical production systems. *Int. J. Adv. Manuf. Technol.* **2020**, *111*, 609–626. [CrossRef]
7. Bartz-Beielstein, T.; Zaefferer, M.; Mersmann, O. (Eds.) Tuning: Methodology. In *Hyperparameter Tuning for Machine and Deep Learning with R-A Practical Guide*; Springer: Singapore 2022;
8. Montiel, J.; Halford, M.; Mastelini, S.M.; Bolmier, G.; Sourty, R.; Vaysse, R.; Zouitine, A.; Gomes, H.M.; Read, J.; Abdessalem, T.; et al. River: Machine learning for streaming data in Python. *J. Mach. Learn. Res.* **2021**, *22*, 1–8.
9. Bifet, A.; Gavalda, R.; Holmes, G.; Pfahringer, B. *Machine Learning for Data Streams with Practical Examples in MOA*; MIT Press: Cambridge, MA, USA, 2018. Available online: https://moa.cms.waikato.ac.nz/book (accessed on 10 October 2023).
10. Losing, V.; Hammer, B.; Wersing, H. Incremental on-line learning: A review and comparison of state of the art algorithms. *Neurocomputing* **2018**, *275*, 1261–1274. [CrossRef]
11. Domingos, P.M.; Hulten, G. Mining high-speed data streams. In Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Boston, MA, USA, 20–23 August 2000; pp. 71–80. [CrossRef]
12. Bifet, A.; Gavalda, R. Adaptive learning from evolving data streams. In Proceedings of the Advances in Intelligent Data Analysis VIII: 8th International Symposium on Intelligent Data Analysis, IDA 2009, Lyon, France, 31 August–2 September 2009; pp. 249–260.
13. Bifet, A.; Gavalda, R. Learning from time-changing data with adaptive windowing. In Proceedings of the 2007 SIAM International Conference on Data Mining, Minneapolis, MN, USA, 26–28 April 2007; pp. 443–448.
14. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.

15. Meignan, D.; Knust, S.; Frayet, J.M.; Pesant, G.; Gaud, N. A Review and Taxonomy of Interactive Optimization Methods in Operations Research. *ACM Trans. Interact. Intell. Syst.* **2015**, *5*, 1–43. [CrossRef]
16. Bartz-Beielstein, T.; Branke, J.; Mehnen, J.; Mersmann, O. Evolutionary Algorithms. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* **2014**, *4*, 178–195. [CrossRef]
17. Lewis, R.M.; Torczon, V.; Trosset, M.W. Direct search methods: Then and now. *J. Comput. Appl. Math.* **2000**, *124*, 191–207. [CrossRef]
18. Li, L.; Jamieson, K.; DeSalvo, G.; Rostamizadeh, A.; Talwalkar, A. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *arXiv* **2016**, arXiv:1603.06560.
19. Gramacy, R.B. *Surrogates: Gaussian Process Modeling, Design, and Optimization for the Applied Sciences*; CRC Press: Boca Raton, FL, USA, 2020.
20. Storn, R.; Price, K. Differential Evolution—A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *J. Glob. Optim.* **1997**, *11*, 341–359. [CrossRef]
21. Senn, S. Determining the Sample Size. In *Statistical Issues in Drug Development*; John Wiley and Sons, Ltd.: Hoboken, NJ, USA, 2021; Chapter 13, pp. 241–264. . [CrossRef]