Noname manuscript No.
(will be inserted by the editor)

Predictions-on-Chip: Model-based Training and Automated Deployment of Machine Learning Models at Runtime

For Multi-Disciplinary Design and Operation of Gas Turbines

Sebastian Pilarski · Martin Staniszewski · Matthew Bryan · Frederic Villeneuve · Dániel Varró

Received: date / Accepted: date

Abstract The design of gas turbines is a challenging area of cyber-physical systems where complex model-based simulations across multiple disciplines (e.g. performance, aerothermal) drive the design process. As a result, a continuously increasing amount of data is derived during system design. Finding new insights in such data by exploiting various machine learning (ML) techniques is a promising industrial trend since better predictions based on real data result in substantial product quality improvements and cost reduction.

This paper presents a method that generates data from multi-paradigm simulation tools, develops and trains ML models for prediction, and deploys such prediction models into an active control system operating at runtime with limited computational power. We explore the replacement of existing traditional prediction modules with ML counterparts with different architectures. We validate the effectiveness of various ML models in the context of three (real) gas turbine bearings using over 150,000 data points for training, validation, and testing. We introduce code generation techniques for automated deployment of neural network models to industrial off-the-shelf programmable logic controllers.

S. Pilarski
McGill University
E-mail: sebastian.pilarski@mail.mcgill.ca

M. Staniszewski
Siemens Canada Ltd.
E-mail: martin.staniszewski@siemens.com

M. Bryan
Siemens Energy Inc.
E-mail: matthew.bryan@siemens.com

F. Villeneuve
Siemens Energy Inc.
E-mail: frederic.villeneuve@siemens.com

D. Varró
McGill University
E-mail: daniel.varro@mcgill.ca

Keywords Prediction-at-runtime · Machine learning · Neural networks · Automated deployment · Code generation · Gas turbine engines

1 Introduction

A cyber-physical system (CPS) needs to (i) autonomously perceive its operational context, (ii) adapt to changes in an open, heterogeneous and distributed environment with a massive number of nodes, (iii) dynamically acquire available resources and aggregate services in order to make real-time decisions, and (iv) continuously provide critical services in a safe, resilient and trustworthy way [10,25].

Gas turbine development is a challenging area of CPSs where complex model-based physical simulations across multiple disciplines (e.g. performance, aerothermal, secondary air system) drive the engineering process at design-time to predict relevant system parameters. However, some of these predictions need to be carried out as part of the real engine control system where the programmable logic controller (PLC) hardware has very limited computing capabilities. Since design-time simulations are computationally very expensive, dedicated runtime predictor programs need to be developed and maintained which may significantly lack in precision compared to their design-time counterparts. In this paper, we seek to leverage advanced machine learning (ML) techniques for such runtime predictor programs.

Research Challenges The design of safety-critical CPSs, such as gas turbines, frequently relies on systems engineering principles facilitating the use of well-defined components interacting with each other via precise interfaces. These components are then deployed to a dedicated hardware platform.

While in the last decade, advances in ML techniques have revolutionized various industrial domains, their use in

safety-critical applications is still limited. In such domains, a certification process may prescribe an extra level of scrutiny, but existing quality assurance techniques for ML fail to *justifiably* ensure safe operation. In fact, *incremental re-certification* is a major industrial driver which aims to scrutinize only those system components which are potentially affected by a change. As a result, substantial certification costs could be saved. However, when the change of a component involves the use of ML techniques, very few guidelines are available for engineers for certification.

Therefore, in the current paper, we investigate three research questions related to introducing ML components on the system level as a replacement of existing components design with traditional engineering methods in the context of gas turbine design as a representative safety-critical CPS.

RQ1 How effective is it to replace an individual (traditional) prediction component or a chain of components (module) with ML-driven counterparts?

RQ2 How effective are different combinations of ML prediction components for engineering models of gas turbines?

RQ3 How to automate the deployment of a trained ML model to a CPS hardware platform to improve maintainability?

Objectives and Contributions In this paper, we aim to exploit ML techniques to enhance the precision and automate the development and deployment of runtime prediction programs. For this purpose, we propose a novel family of predictors-on-chip by (1) training various ML models using design-time simulation data and then (2) automatically deploying the trained ML models to the production environment by automated code generation. Design-time physical simulations (and potentially existing field data) provide high-quality data for the training of a ML model, and the trained model is deployed without further alterations to the runtime system to reduce (software) engineering efforts.

The main novel contributions of the paper can be summarized as follows:

1. We present a *novel industry application of existing ML techniques in the context of gas turbine design* where the penetration of ML techniques is still sparse.
2. Given the safety-critical nature of gas turbines as a CPS, we propose various deviations from common ML best practices:
   (a) We use only 20% of data for training and validation and 80% for testing to ensure generalized behaviour.
   (b) In addition to traditional error metrics which evaluate the average behavior of ML components, we also *investigate worst-case behaviour* and *the direction of error* (e.g. when under-estimating a parameter can be more problematic than over-estimating it).
   (c) We *incorporate deployment constraints* imposed by the target hardware platform *as design consideration* for the ML component.
3. We carry out an extensive experimental evaluation of various ML techniques taking a system-level perspective in the context of gas turbine design for bearing load prediction with key insights.
   – ML-driven components consistently provide better bearing load prediction than existing (traditional) predictors. However, the errors of individual components may accumulate in a component chain thus violating compositionality principles (RQ1).
   – Replacing multiple components with a combination of ML prediction components improves upon existing traditional prediction modules, but replacing a chain of traditional components with a single ML component may be even more beneficial (RQ2).
4. We combine ML and code generation to automate the deployment of ML components to programmable logic controllers (PLCs) (RQ3).

The paper primarily focuses on gas turbine design as a motivating CPS scenario of high industrial relevance, and our contributions are evaluated in this context. However, we believe that many of our core ideas could be adapted to other CPS domains where high-fidelity runtime predictions are necessitated in a real-time execution environment.

Structure of the paper. After an overview of related work (Section 2), we discuss in Section 3 how ML components can be used as functional prediction components in a CPS. Furthermore, we summarize core ML concepts used in the paper. We provide a brief overview of some engineering challenges of gas turbines (Section 4). Then, we propose a high-level architecture Section 5 for integrating ML techniques and components into the design of gas turbines. Subsequently, we present the prediction module problem definition in Section 6. We propose load prediction module architectures in Section 7 along with how to use ML best practices to train various ML predictors using existing multi-disciplinary simulators of gas turbine design as data sources. We evaluate and compare such ML-based runtime predictors with existing runtime predictors as individual components as well as in prediction chains (where the output of one predictor serves as the input of another predictor) (Section 8). Next, we propose an automated code generation-based technique to deploy an ML model to a real controller chip with limited computing resources used in the real-time control system of existing gas turbines (Section 9). We discuss threats to validity (Section 10) and finally, Section 11 concludes our paper.

2 Related Work

While machine learning, systems engineering, and runtime CPS research areas are quite mature, the intersection of these

disciplines remains at an early stage of research. We categorize related literature into several main areas: artificial intelligence (AI) techniques in gas turbine design, parameter prediction, digital twins, model-driven engineering and ML, and code generation for PLCs.

Load and Bearing Prediction Many applications of ML and AI techniques exist to various parameter prediction problems in a larger engineering context (excluding gas turbine design). For example, there exists a wealth of literature for load prediction in buildings. Unsupervised and supervised deep learning techniques for cooling load prediction are evaluated in [13]. The authors of [43] explored 12 different prediction algorithms for building load prediction. A neural network steam load predictive model was developed in [21] while a support vector machine algorithm was used in [26] to predict hourly building cooling load and evaluated in comparison to back-propagation neural networks. Suspended sediment load in river water was predicted in [23] using support vector machines and neural networks.

There also exists literature which relates to bearings in various contexts. Several ML algorithms trained on high resolution bearing fault simulations are evaluated in [38]. Support vector machines and relevance vector machines classifiers are created in [44] for fault diagnosis and classification based on testing rig data. Feature models for classification of bearing faults are created in [36].

Our paper exploits similar ML techniques for gas turbine design and expands upon them by (1) evaluating various system-level architectures, (2) assessing design decisions on ML algorithms and feature selection, and (3) proposing an automated technique to deploy a trained ML model to a dedicated (resource-constrained) production platform.

AI in Gas Turbine Design There exists significant literature concerning ML and AI techniques applied to gas turbine design. For example, a hybrid AI and numerical approximation methodology was used to produce new turbine designs in [15], and areas of gas turbine design where AI could be applied described in [32]. Additionally, there exist works on machine learning for runtime prediction: using neural networks for internal cycle parameter prediction [11] [17], fault detection [31], engine sensor and component fault and health diagnosis [18], and operating parameter prediction [24]. A number of works also study neural networks for prognosis [14] [19] and propose architectures such as physics-informed recurrent network cells [29].

Our work can be regarded as a novel case study for intelligent gas turbine design with the novel aspects of (1) contextualizing our work to systems engineering and CPSs, (2) studying the impact of introducing ML-based predictor modules into an existing system-level architecture, and (3) directly deploying ML models into the control system.

ML in MDE/Digital Twins The intersection of model-driven engineering (MDE) and ML is a rapidly growing research area. Within MDE literature, there are works applying ML for metamodel classification [30], model transformation [9], artificial intelligence for requirements aware runtime models [5], and code generation [34]. Additionally, some publications focus on domain specific languages for machine learning and code generation [8] [22].

Digital twins are a sprouting research area at the intersection of model-driven software and systems engineering, telecommunication and artificial intelligence [28, 7]. A digital twin prediction model for tool wear and condition is presented in [35]. A framework for developing ML models from a digital twin perspective is proposed and validated on a robot interacting with external parts in [4]. The authors of [27] developed a digital twin framework and applied it to fault diagnosis and prediction.

Our work presents new perspectives and problems in these intersecting research areas. We present methodology for applying MDE principles for automated deployment of ML algorithms. Existing works focused on using ML for MDE, and generating high level framework code implementations of algorithms (Tensorflow, PyTorch, etc.). Meanwhile, we develop a code generator for machine learning in off-the-shelf, low-level PLCs, where hardware limitations imposed by industrial environment precludes porting standard frameworks or libraries. Likewise, our work presents deployment of internal system behaviour prediction and monitoring within the low-level control system rather than to external (more computationally powerful) machines maintaining digital twins. We present the efficacy and possible risks of our methodology.

Code Generation for PLCs Literature exists related to programmable logic controller code generation. This includes control loop code generation from defined state automata [37] [40], ontologies [39] [41], and formal specification languages [12]. [16] presents model predictive control on a PLC.

Our work extends previous PLC code generation capabilities to include ML such as neural networks in a manner which enables MIMO model predictive control despite the limited hardware capabilities.

3 Machine Learning in Systems Engineering

3.1 Runtime Predictions in Cyber-Physical Systems

Most cyber-physical systems have a well-defined system architecture using functional components where the role of each component is to (periodically) calculate a function as output when a given set of inputs is provided (see Figure 1). Model-based systems engineering aims to determine necessary components, define input and output requirements for

each component, and connect all components together. Design decisions for components and component interactions result from domain knowledge and experience.

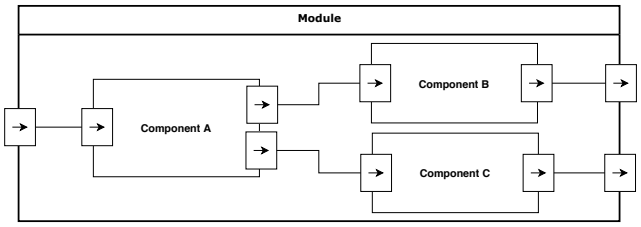


Fig. 1 Functional architecture with components.

Definition 1 (Functional component) A functional component $Comp = (InP, OutP, Fun)$ consists of a set of input ports InP and output ports $OutP$ in order to provide a certain functionality Fun.

Definition 2 (Functional module) A functional module $Mod = (InP, OutP, Comp, Link)$ (aka compound component) consists of input ports InP and output ports $OutP$, a set of (simple or compound) components $Comp$ with links $Link$ where each $l \in Link$ connects (1) an output port of component c_i to an input port of component c_j, (2) an input port of module Mod to an input port of internal component c_i and (3) an output port of internal component c_j to an output port of module Mod.

Before deployment to production, such designs must undergo thorough design review and certification, which involves (multi-disciplinary) simulations to investigate and predict various characteristics (e.g. thermal, stress, reliability, performance, etc.). In case of high fidelity models, such simulations provide precise estimates, but they need to be executed on powerful server farms, yet a single simulation run may still take hours to complete.

Definition 3 (Prediction module) A prediction module $Pred = f(In, Mod) : Out$ created in accordance with functional module Mod operates at runtime by calculating output data Out for each output port $Mod.OutP$ when input data In is provided to each input port $Mod.InP$ of the respective functional module.

Many of such predictions need to be deployed as part of the CPS in operation executing on dedicated target hardware within the control loop. However, due to high computational complexity, most existing design-time simulators cannot be deployed to a runtime environment, thus custom prediction components need to be developed. As a consequence, the precision of such runtime prediction components may be lagging significantly behind their design-time counterparts.

In this paper, we aim to exploit various ML techniques to train and deploy prediction components to CPSs in operation. Our approach uses design-time simulation results for training various ML models, which are then deployable to the target hardware thanks to their reduced memory footprint and small number of calculations. While at design-time, the prediction of an ML module may be less precise than a prediction obtained by using simulators, *at runtime, the prediction of a ML module can be more accurate than existing traditional prediction modules*.

3.2 Core Machine Learning Concepts

Machine learning (ML) [6] is the research discipline focusing on algorithms which learn to make inferences and predictions from data without explicit logical instructions. Machine learning usually relies on heavily statistical, probabilistic, and derivative-based algorithms.

A major subclass of ML approaches are *supervised ML techniques*, which will be used exclusively in the current paper. Supervised algorithms require both input and corresponding target output provided and seek to learn a function which transforms the input to the target output. Applicable tasks can be either classification (outputs belong to a defined discrete set of prediction classes) or regression (outputs are predicted along continuous variable spaces).

ML architecture There exist a multitude of ML algorithms with vastly different underlying techniques. However, there exist some general concepts relevant to the majority of those.

In a supervised learning context, all algorithms seek to minimize a *loss function*. Some algorithms are derivative-based (learn by gradient adjustment) and thereby require the loss function to be differentiable. The actual learning in most supervised algorithms learn occurs by repeatedly adjusting model parameters such as weights or probabilities in order to minimize the loss function following some optimization strategies. Examples of supervised ML algorithms include neural networks, logistic regression, support vector machines, Bayesian approaches, etc.

Hyperparameters Many algorithms have extra parameters, typically designated as hyperparameters, which cannot be estimated from the data and require manual adjustment during the training process. A sample hyperparameter can be the number of neurons or layers in a neural network.

Data sets Machine learning heavily relies on the availability of high quality data, which is typically separated into three independent sets: training, validation, and test. The *training set* is used (at design-time) to train or fit ML algorithms.

However, as these algorithms can be sensitive to perturbations in data, models must be tested on additional independent data sets. The effect of tuning hyperparameters is evaluated on the *validation set*. The best tuning of an algorithm, as determined on the validation set, is then tested on a final independent *test set*.

Algorithms may not perform equally on each data set. An algorithm suffers from *overfitting* when it achieves high performance on a training set, but not on the test set, which means that the algorithm does not properly generalize. An algorithm suffers from *underfitting* when it does not achieve sufficient performance on any of its data sets.

Specific attention may be required to *decrease the number of inputs* for an algorithm, as high dimensionality may result in prohibitively high amount of data or training time.

3.3 Pipeline of Machine Learning

The engineering of a ML algorithm follows a general pipeline.

1. **Problem definition:** Clearly define objectives, requirements, and evaluation criteria for the problem.
2. **Collect data:** Ascertain that data is high quality, relevant and (ideally) abundant.
3. **Process data:** Create a clean data set from the collected raw data (distillation). This step could involve the removal of certain pieces of data with errors and the removal of extraneous parts. Data is split into appropriate test, validation, and test sets.
4. **Feature engineering:** Determine which inputs and outputs (features) are best suited for the defined problem. May also involve combining data to form new features not present in data collection.
5. **Develop ML architecture:** Define and create a ML architecture for the problem (e.g. use linear model or neural network with a given architecture, etc.).
6. **ML training:** Train or fit the ML architecture using the training set.
7. **ML validation:** Validate the performance of the ML architecture with a given set of hyperparameters on the validation set.
8. **ML testing:** Test the ML architecture (with best determined hyperparameters) on the test set.
9. **Optimize ML for deployment:** Optimize the ML for deployment onto the target production platform.
10. **ML in runtime:** Run the ML component on the real production platform.

3.4 Categorizing Machine Learning Models

The machine learning community often refers to machine learning algorithms as models. To avoid ambiguity, we will refer to them as ML models. From a systems engineering perspective, models may exist at both design time and runtime. To place ML models in a systems engineering context, ML models will also be differentiated accordingly.

- **Pre-ML models:** A pre-ML model is characterized by the creation of a problem definition and datasets. These are required for ML as a starting point.
- **Design-time ML models:** Such models are used by engineers while the system is still under development, dominantly for training and validation purposes, which are carried out using powerful dedicated hardware (e.g. GPUs).
  - **Training ML Model:** A training model includes training data and an ML architecture and a specific combination of hyperparameters. Typically, a wide range of training models are trained as part of tuning step.
  - **Validated ML Model:** A validated model has undergone significant training and tuning, thus it represents the best-performing training model for a given architecture on the validation set. Its performance is typically verified on an independent test set. Such predictors (for each ML architecture) are considered trained and finalized.
- **Runtime ML Models:** Runtime models are deployed on the real system in operation as runtime predictors. As such, they need to operate on the target computing platform (e.g. on a controller chip) to process real inputs and provide real-time output (as part of a controller loop).
  - **Deployment ML Model:** This model evolves from a validated model which has been optimized to fulfill production requirements (memory limits, computation time, etc.). It is ready for production testing.
  - **Production ML Model:** This model is a deployment model actively running in production on the designated target hardware.

Runtime ML models can also be separated into two types based on the exploitation of runtime data:

- **Adaptive:** An adaptive model participates in online learning during production. This model must remain *trainable* and must maintain relevant learning settings for its problem. This may involve keeping some of its training model configurations, but removing others. Note, the model should be optimized to run and learn on its production platform.
- **Nonadaptive:** A nonadaptive model does not actively learn during production. The model can be considered *compiled*, for a static instruction set of mathematical operations is simply optimized to run on the production platform. The majority of training model settings (dropout configurations, optimizer function, etc.) can be excluded from this model to reduce memory footprint.

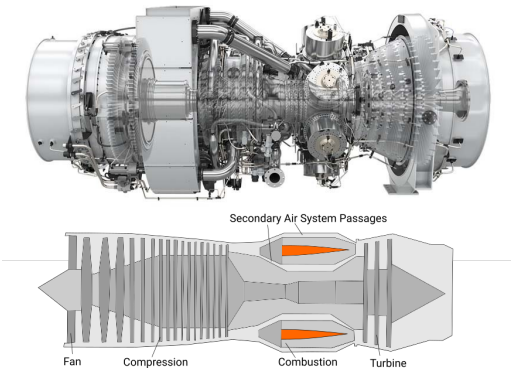


Fig. 2 Example gas turbine [2] with simplified, labeled model.

4 Engineering Gas Turbines: An Overview

Our approach and methodology arises from engineering challenges faced by Siemens Power and Gas. First, we provide an overview of the engineering context.

4.1 Engineering Context

Gas turbines (Figure 2) are a common type of internal combustion engine used to generate power. Due to a multitude of complex physics interactions (fluid interactions and combustion), thorough design validation is required to ascertain that the control system properly regulates engine behaviour to prevent failures. An event which can trigger a damaging engine failure is an over-load or under-load of a bearing. As a result, significant engineering efforts are spent at design time to ensure the engine maintains appropriate load on the bearings at all times.

Design time Some engine models integrate an active bearing load control component managed by the control system. For such a component, the control system expects the current load on the bearing and the target load as input. Unfortunately, no sensors exist to measure the actual bearing load, thus it must be estimated from other sensor information and engine parameters. In current engineering practice, engineers need to manually create predictors which estimate the bearing load, which is a very time consuming process.

Due to high costs of manufacturing and engine testing, the majority of design validation is performed by running engineering simulations (mainly physics-based equation solvers for simulated conditions on physics engineering models). Such simulations help engineers define safety envelopes, i.e. parameters within which the engine can safely run.

Engineers run simulation tools and manually extract relevant data from the results, which is then used to determine correlated parameters to form the basis of generated prediction functions. These functions are then integrated into load prediction modules in the control system.

Runtime At runtime, the engine system runs the control system code. Sensor readings are forwarded to the load prediction module, which uses the engineer-developed correlation functions to predict what the current bearing load is. Using this prediction, the control system controls actuators to adjust the active bearing load control component to shift the current bearing load towards a defined target load.

4.2 Simulation of Physical Engineering Models

The design of gas turbines is carried out with many teams along several disciplines which include whole engine, secondary air system, aerothermal, thermo-mechanical, etc. Teams develop models for each system component within their discipline and, at one stage, each team's designs must unified for the entire engine as a whole.

Engineering models are developed within engineering tools and are heavily used during the design and validation of a model. Models are changed to test new design ideas and simulate how they perform. Physical simulations are run on models which involve solving of physics equations (flow, force, etc.) across the model. The majority of the engineering simulations use iterative physics solvers which attempt to converge for each parameter in the model to a set threshold. For a given model, a convergent solution, cannot always be found by the simulation solver. Simulations are evaluated across the range of engine operating conditions.

Engine disciplines, of course, interact and depend upon each other. Thus, periodically, models must be converged. Due to model interdependence, each discipline model contains input parameters from other models. Such parameters remain unchanged until changes are approved from the other disciplines. To converge models, simulations are run in a loop, until model parameters cease changing. This requires a loop until convergence, as models affect each other. Each small parameter change within a model's simulation run affects the inputs of other models, which, in turn, will affect the model. This is illustrated in Figure 3.

In the scope of this work, we incorporate three relevant gas turbine engineering disciplines, namely, *performance*, *aerothermal*, and *secondary air system*.

4.2.1 Performance Discipline

Modeling and analysis of whole engine behaviour is the basis of engine design and enables decisions on which components engineers should focus on to best improve engine operation. Whole engine modeling is the focus of the performance discipline, and primarily uses a 1D thermo-dynamic meaning direction is limited to one spatial dimension.

This model maintains a static directed graph, with each node relating to an engine component, station, or even on-engine sensor. Edges connect various engine components to-

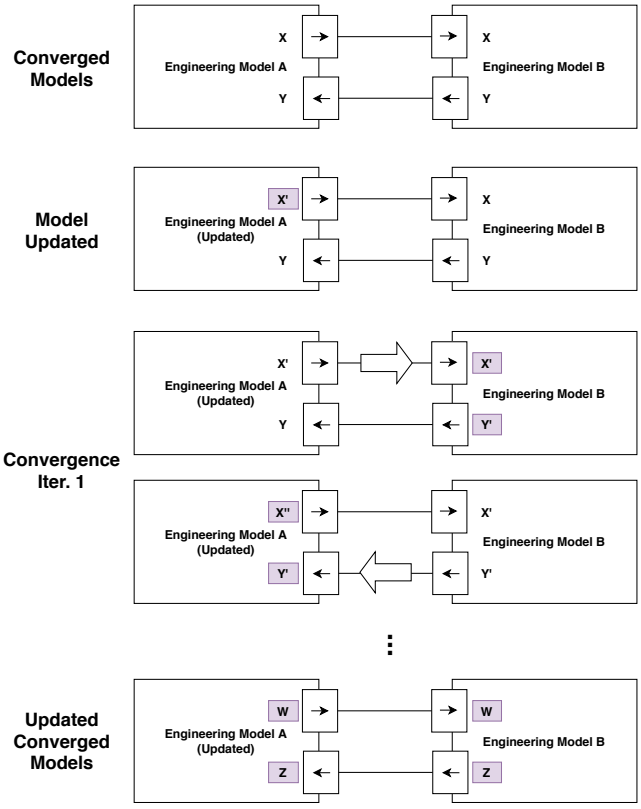


Fig. 3 Multi engineering model convergence illustrated. Purple highlights changes.

gether and define flow paths. The model requires the abstraction of components (blades, compressors, etc.) into simple factors (efficiency, capacity, etc.) which form the attributes of the graph nodes.

Definition 4 (Directed attribute graph) A directed and attributed graph $DAG = (N, E, A, src, trg, attr)$ has a set of nodes N, a set of edges E connected to source and target nodes with respective functions $src : E \mapsto N$ and $trg : E \mapsto N$. Extra attribute values can be assigned to nodes and edges using a predefined set of attributes A as $attr : (N \cup E) \times A \mapsto \Re$.

Definition 5 (Engineering performance model) An engineering performance model is a directed attributed graph $EPM = (N, E, A, src, trg, attr)$ with defined engine components and stations as nodes N, flow paths between components as edges E with sources src and targets trg, and attributes $attr$ such as pressures, flows, efficiencies, etc.

The engineering performance model allows an engineer to run the engine under substantially different operating conditions, using limits (for example CO, turbine temperature, turbine speed, NO_x, etc.), and other settings. During simulation, an iterative convergence algorithm is executed to find the behaviour at each operating point (temperatures, pressures, flows, rotation speeds, etc.).

Definition 6 (Engineering simulation) An engineering simulation, $ESIM = (EM, OP, SOLV)$, involves setting certain operating conditions (attributes) OP for an engineering model EM and running a physics-based solver, $SOLV$. The solver calculates the behaviour across the entire engineering model as a result of the given operating conditions.

These determined behaviours are highly important for control loops during the runtime of the engine as they provide a theoretical baseline for what should be occurring across the whole engine. However, as performance simulations are computationally too complex to run within the engine control system, some of the behaviour calculations are approximated using runtime predictors.

In this paper, we use the performance model to create an array of operating points (varying temperature, humidity, engine power output, etc.), which will generate the on-engine sensor readings for each point. Then these values will serve as the input to a ML-based predictor of bearing loads.

4.2.2 Aerothermal Discipline

Engines are designed to rotate turbine blades as efficiently and quickly as possible to generate power. The aerothermal discipline aims to design compressor and turbine blades.

Designing the airfoil and its cross sections of a turbine blade involves the use of 2D and 3D models to balance the aerodynamic properties of the blades with lift. Such simulations involve solving Bernoulli equations to determine stresses, lift, efficiencies, the required amounts of flow, temperatures, and pressures for the turbine and compressor blades to function properly. Aerothermal flow requirements are used as input to run secondary air system models under correct assumptions.

4.2.3 Secondary Air System Discipline

Gas turbines endure high temperatures and pressures as a result of compression and combustion. Thereby efficient cooling is paramount to maintain proper engine function, structural integrity, and significant engine life. The secondary air system seeps air from the main intake path and guides air through secondary passages to cool parts of the engine, primarily discs, as well as to balance pressures. These passages are lined with seals and restrictions to meter the flows required in the cooling process.

The vast majority of secondary air system design time is spent working on a 1D model, which is represented as a directed graph. Each graph node (with a set of attributes) corresponds to a component (e.g. inlet, seal, pump) within the secondary air system and graph edges form connections (paths) between components.

Definition 7 (Engineering SAS model) An engineering SAS model $ESM = (N, E, A, src, trg, attr)$ is a directed attributed graph with SAS components (pumps, seals, etc.) N, air flows E with sources src and targets trg, and attributes such as pressures, flows, loads, and temperatures $attr$.

A simulation run involves setting the attributes of the engineering SAS model (e.g. pressures and temperatures from the combustion process), and iteratively solving the physics flow equations for the remaining node attributes until convergence. Pressures, temperatures, and loads are calculated on the different components within the graph.

The SAS system must behave correctly at runtime otherwise the engine will suffer large stresses and high levels of heat. Because each SAS simulation run is a computationally complex operation, certain SAS predictors must exist at runtime to help the control system determine proper behaviour and prevent safety-critical operation errors.

In this paper, we propose to create the data necessary to train predictors for runtime bearing load prediction by running the secondary air system model simulations across a vast number of operating conditions and storing the outputs. The solver-calculated pressures and loads are used to calculate bearing loads.

4.3 Objectives

In order to decrease engineering efforts and better maintain proper bearing loads, the challenge is (1) to improve predictions, and (2) to automate the process of developing and deploying predictor components.

These high-level challenges trigger three main technical challenges in a systems engineering context:

1. **Improve quality of runtime predictions:** Better runtime predictions of bearing loads result in improved maintenance of target loads, improved engine life, and prevention of safety-critical failures.
2. **Automate the design process of predictors:** An automated design process saves engineers significant time which can be better spent on other tasks. An additional benefit is that more data and simulations can be ran in the background yielding better predictors and evaluations.
3. **Automate deployment of runtime predictors to dedicated hardware:** The hardware on which the control system of such engines is developed is a Programmable Logic Controller (PLC). PLCs are primarily designed to be robust, consistent and capable of functioning in any environment. As such, these controllers tend to be weak by modern computation standards, as they are single-threaded, they disallow dynamic allocation, and do not optimize provided code. This forces the deployed predictor to be computationally efficient.

5 System Architecture for Load Prediction

This section presents an architectural overview of the proposed bearing load prediction system for gas turbines (illustrated in the form of a SysML block diagram in Figure 4). The architecture incorporates three stages of development: **design time**, **deployment**, and **production**. It details the process starting from running engineering design tools to physically running a real-time engine bearing load predictor in the control system.

The **design phase** begins after there exist approved versions of performance, aerothermal, and secondary air system design models. These models enable simulations of each of the corresponding subsystems. A mixture of data from all three subsystems is needed to develop the predictor in the Predictor Creator module. The simulations of the various systems are connected as a chain. The chain begins with a performance model being run at an operating condition. Performance outputs are required for aerothermal simulations to be run, and the outputs of both are required by the secondary air system model. A Tool Runner was developed to automate and launch each of the analysis tools across a desired set of operating conditions and to organize simulation output data in a way which maintains consistency between each of the performance, aerothermal, and secondary air system outputs. This data organization is crucial as it enabled data querying used for data distillation. The Distiller filters out important parameters and combines pieces of data to create data sets for machine learning. The distilled data sets are used in the Prediction Creator, in which an ML predictor is developed (i.e. a neural network in our case).

The resulting predictor is deployed onto a PLC in the **deployment phase**. The predictor is saved in a customized JSON format which includes the model type, connections, and weights within the network. The predictor is provided to the Code Generator module which contains a Source Code Generator and Configuration Generator. The former creates PLC source code for the runtime predictor including instructions for activation functions and proper execution order of instructions. The latter packages this source code together as well as generates the configuration requirements such as data structures, and metadata such as versioning. Using these configurations and source code, the predictor is uploaded onto the PLC as a routine.

In **production**, on-engine Sensors are read periodically and their values are delivered to the Load Predictor. The Load Predictor uses these input values and updates the current bearing load predictions. These predictions are used by the Active Load Controller to calculate how to adjust the actuator to increase or decrease the bearing load to the target value from the prediction. A change induced by the actuators causes a change in the bearing load. This loop repeats indefinitely until engine shut down.

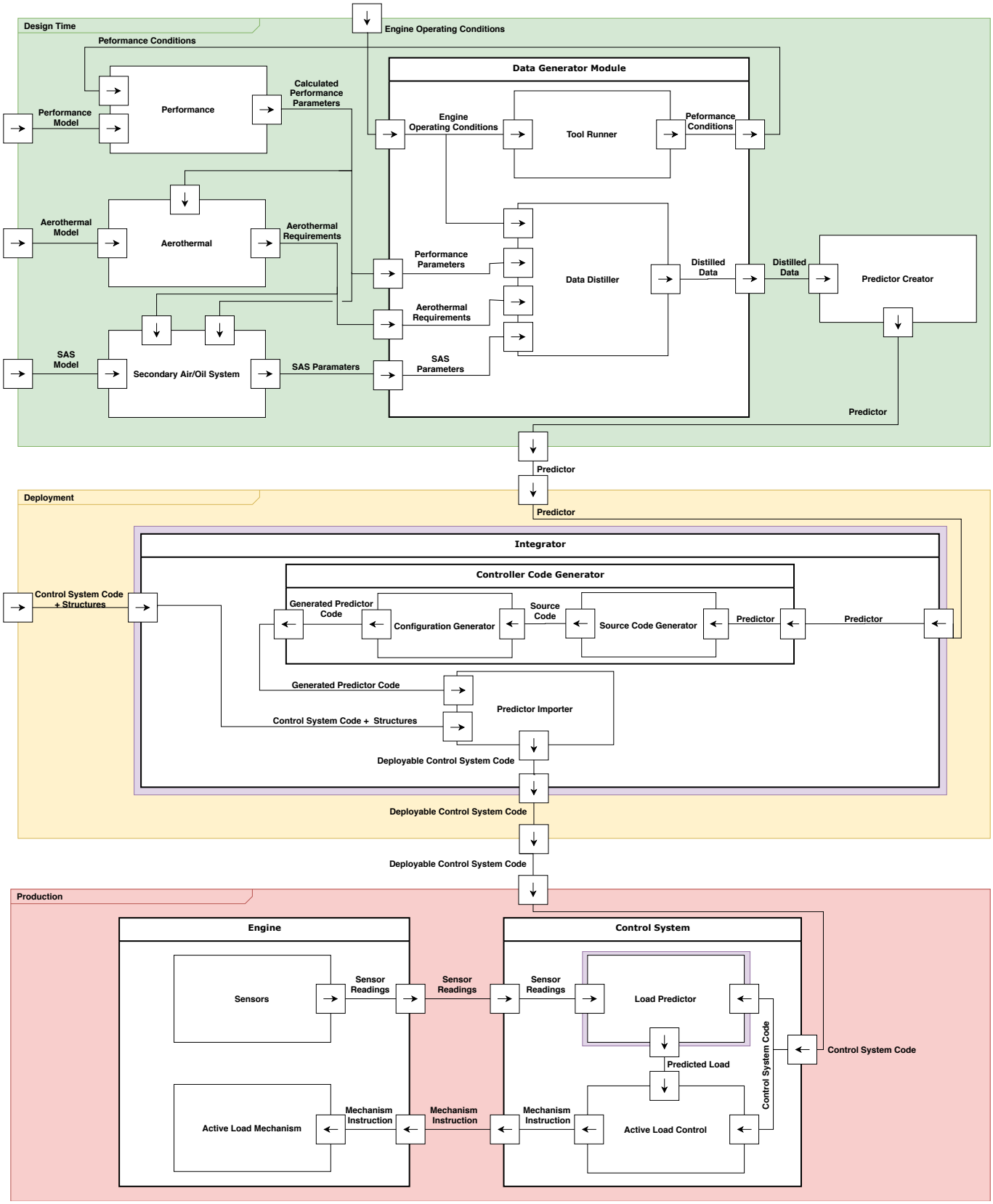


Fig. 4 SysML internal block diagram style overview of entire system from design time to deployment and runtime. Contributions of this paper are highlighted with a purple border.

Industrial Benefits The adoption of this architecture and the technical details presented in the following sections have yielded several notable benefits for Siemens Energy:

1. **Improved predictions:** ML-based prediction modules have demonstrated improvement upon existing traditional prediction modules wrt. pre-defined metrics. For example, mean absolute error was reduced by up to 60x by using ML-based predictions (see Section 8).
2. **Process time saving:** The process of automating data generation and component creation periodically saves 20+ engineering days. As this a repeatedly recurring process, it translates into significant cost reduction.
3. **Versioning:** Due to savings in process time, if engineering models are updated, new, improved, versions of prediction modules can be deployed.
4. **Improved control software:** Thanks to automated code generation, the process of writing, debugging, and testing prediction modules is significantly simplified, simultaneously improving product quality and decreasing costs.
5. **Multi-platform support:** Code generation enables a single ML model artifact to be deployed to multiple production platforms (e.g. different-vendor PLCs).

6 Prediction Module Problem Definition

Below, we define the requirements, evaluation and deployment criteria for the prediction module, and present an existing (traditional) solution for the problem.

6.1 Requirements for the Load Prediction Module

The load prediction module must fulfill certain design requirements (related to inputs, outputs and computational cost of operations) as well as specific deployment constraints in order to operate in the existing engine architecture.

- **Inputs:** Inputs are limited to any subset of the available on-engine sensors. These inputs can be considered as valid - there exist other mechanisms in the engine to test sensor validity - the sensor readings will not be woefully unreliable/inaccurate.
- **Output:** The output of a given load prediction module is a predicted bearing load provided in the standard units employed in the control system. Prediction will take place on three (real) bearings referred to as Bearing 1, Bearing 2, and Bearing 3.
- **Computation budget:** Computation is limited to at most L computational operations (e.g. addition, multiplication, subtraction, reads and writes). This computation limit is imposed by constraints of the target hardware platform on which the predictor is deployed to guarantee in-time prediction (see Section 9 for further details).

The prediction module must be capable of being deployed to a 32-bit programmable logic controller (PLC). The deployed module must adhere to the following constraints:

- **Operations:** The deployed predictors are limited to integer, Boolean, and floating point operations in 32 bit.
- **Allocation:** No dynamic memory allocation is possible. Defined variables (software registers) are typed and persisted at runtime.
- **Deadline:** A prediction must be completed within a deadline t for real-time control behavior.

6.2 Evaluation Criteria

Our experimental evaluation is based on normalized bearing load prediction error (where smaller is better). Each prediction model is evaluated on the following *four metrics* and compared with the existing traditional prediction module presented in Section 6.3:

- **Maximum Overprediction (MO):** Largest difference between predicted load and actual load, where predicted load is greater than actual load.
- **Maximum Underprediction (MU):** Largest difference between predicted load and actual load, where predicted load is less than actual load.
- **Mean Absolute Error (MAE):** The average of the absolute value between predicted load and actual load.
- **Root Mean Square Error (RMSE):** The square root of the average of the value between predicted load and actual load squared.

MAE and RMSE are standard ML evaluation criteria and help understand the average performance for a given predictor. MO and MU are motivated by the engineering context where worst case scenarios are highly relevant. Bearing failures can be safety-critical when a bearing is overloaded (result of underprediction) or underloaded (result of overprediction). Thus it is imperative to evaluate how well a prediction module behaves at the extremes. As such, we intentionally deviate from ML best practices to adapt them to a safety-critical engineering context.

6.3 Existing Load Prediction Module

The existing (baseline) load predictor module is a sequential composition of two prediction components (Figure 5). Our baseline prediction architecture represents the current engineering best practice at Siemens for analyzing gas turbines.

- The first component is a correlation map which correlates (a subset of) on-engine sensor readings to some unmeasured internal engine parameters such as pressure. These internal parameters exist as attributes in the *performance discipline's* 2D thermo-mechanical model.

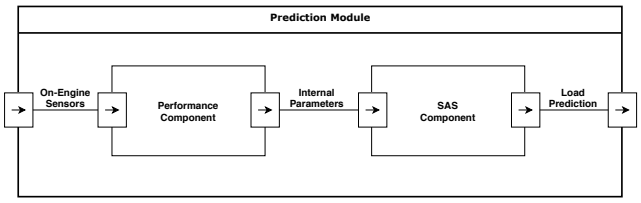


Fig. 5 Prediction module with components.

– The second component is a correlation map from these internal engine parameters to the bearing loads. The bearing loads were determined via calculations obtained from the 2D *secondary air system model*.

Each prediction component was developed individually with *performance discipline* engineers developing the first, and *secondary air system discipline* engineers developing the second. These two components were then combined to form the load predictor module. The two components, for clarity, are henceforth referred to as *performance component* and *SAS component*.

The internal engine parameters (output of performance output and input to SAS component) have been traditionally used in the prediction chain due to the strong physics interlinkage between them and the bearing loads.

7 ML-Based Load Prediction Architectures

This section proposes several ML-based load prediction module architectures together with the underlying ML models.

7.1 Proposed Prediction Module Architectures

We propose a number of purely ML-based and hybrid architectures for the prediction module (visualized in Figure 6):

1. **1ML:** *Single ML model:* A module with this architecture will be limited to one ML model component which predicts bearing loads directly from on-engine sensors.
2. **HSML:** *Hybrid model with ML SAS component:* Such a module keeps the existing performance component and replaces the existing SAS component with an ML model.
3. **HPML:** *Hybrid model with ML performance component:* This architecture will replace the existing performance component with an ML model and keep the existing SAS component.
4. **2ML:** *Dual ML model:* Both existing components will be individually replaced in this architecture.

We propose these architectures to study the replacement of modules and components in the existing context of gas turbine design. These architectures cover all permutations of the prediction chain in the existing module.

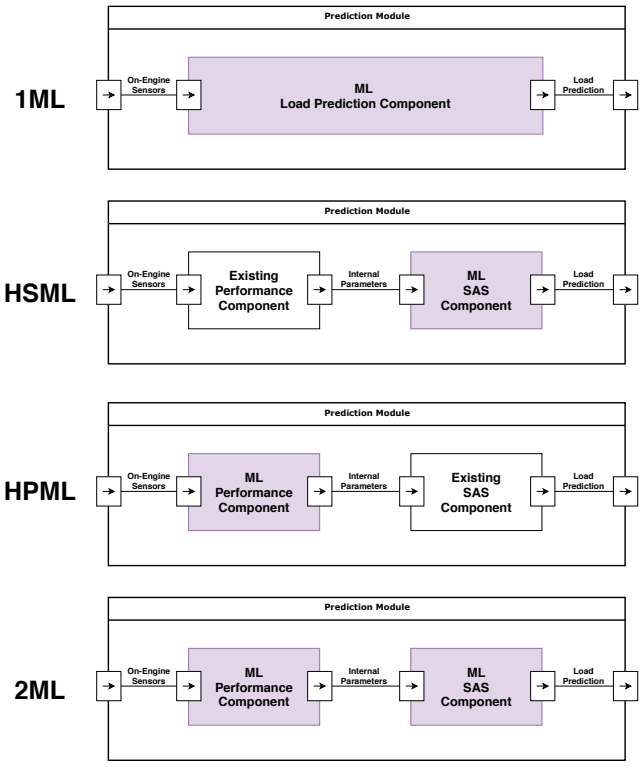


Fig. 6 Architectures of investigated ML-based predictor modules.

Each architecture will be instantiated with two sets of starting sensor inputs designated as *limited* and *all*. The *limited* set includes only sensors used as input to the existing traditional prediction module. The *all* set will enable the prediction module to access all available on-engine sensors. Modules with limited sensors will be tagged as and modules with access to all sensors as **(all)**.

For each architecture, we experiment with two subclasses of ML techniques using *Bayesian ridge regression* and *feed-forward neural network* described in Section 7.2. In the 2ML architecture, we study the effect of different combinations of underlying ML models.

7.2 Background on ML Architectures

Two ML architectures are explored in this paper: (1) Bayesian ridge regression and (2) feed-forward neural networks (NNs).

– Bayesian ridge regression was selected due to its low computation requirements (which is beneficial for deployment to PLCs) and effectiveness in various application contexts (e.g. Bayesian approach fits to existing data well).
– Neural networks have demonstrated to successfully learn complex functions from a variety of data sets. Extremely large neural networks and deep learning techniques are not explored due to the hardware limitations of the underlying PLC controller.

While we also studied and experimented with other ML architectures, we restrict our presentation to these two techniques in the paper to limit the number of combinations of hybrid components.

We provide formal definitions for the core ML models used in the context of the paper.

Definition 8 (ML model) A *machine learning (ML) model* $ML = (\mathbf{x}, \hat{\mathbf{y}}, Data, Arch, HP)$ consists of a vector of inputs $\mathbf{x}$, a vector of outputs $\hat{\mathbf{y}}$, a (training) data set $Data$ with pairs of associated input and output values $(\mathbf{x_i^{(d)}}, \mathbf{y_i^{(d)}})$, an ML architecture $Arch$ and a set of hyperparameters HP.

During *prediction*, the model applies a function f_{ML} to the input to predict the output i.e. $\hat{\mathbf{y}} = f_{ML}(\mathbf{x})$. The function is determined during training by minimizing a given loss function based on training data. An architecture $Arch$ implicitly includes a *loss function* $loss(\hat{\mathbf{y}}, \mathbf{y}) = val$ which calculates a metric value val, measuring how different a set of predictions $\hat{\mathbf{y}}$ is from its corresponding set of actual values $\mathbf{y}$. Common loss function metrics include mean square error and cross entropy loss [6].

7.2.1 Bayesian Ridge Regression

Bayesian ridge regression is a linear ML model, which predicts only a single value as output.

Definition 9 (Linear ML Model) A linear ML model $LIN = (\mathbf{x}, \hat{y}, Data, Arch, HP)$ has an architecture $Arch$ with a (single) output prediction defined as a weighted sum ($\hat{\mathbf{y}} = f_{LIN}(\mathbf{x}) := \sum_i w_i \cdot \mathbf{x}_i$) of the input variables.

Weights w_i within a linear model are determined by some algorithms which minimize a defined loss function on training data. Due to the simplicity of linear models, they are fast to train and compute, and they have low storage needs for parameters (one weight value for each input value). An additional benefit of their simplicity is that they rarely overfit: they have very few parameters and must learn to fit the overall trend of data. On the other hand, they may not properly learn to handle outliers well.

There are many techniques to determining suitable weights. Bayesian ridge regression determines a linear ML model using Bayesian inference. This linear approach is more robust to outliers and it can also provide confidence level in prediction as a measure of variance from previous data.

Definition 10 (Bayesian Ridge Regression [42]) Bayesian ridge regression $BR = (\mathbf{x}, \hat{y}, Data, Arch, HP)$ is a linear ML model whose architecture relies on Bayesian inference to determine the weights under two assumptions:

1. The prediction has a probability density which is normally distributed $p(\hat{y}|\mathbf{x}, \mathbf{w}, \alpha) = \mathcal{N}(\mathbf{y}|\mu, \sigma^2)$, where $\mu = \mathbf{w}^T\mathbf{x}$ and $\sigma^2 = \alpha$ is some internal parameter.
2. A prior on the weights is normally distributed with a mean of 0, i.e. $p(w|\lambda) = \mathcal{N}(w|0, \lambda^{-1}I_p)$, where λ is an internal parameter related to precision.

Formally, the output of the Bayesian ridge regression model is a (normal) distribution, which is then turned into a single value prediction $\hat{y}$ by taking its mean. An iterative learning algorithm jointly estimates the parameters, w, α, and λ where α and λ are estimated by maximizing the log marginal likelihood.

7.2.2 Neural Networks

Neural networks are composed of artificial neurons which apply an activation function to their inputs to get the output.

Definition 11 (Artificial Neuron) An artificial neuron $an = (\mathbf{x}, y, Bias, f_a)$ consists of a vector of inputs $\mathbf{x}$, a single output y, a bias term $Bias$ and activation function f_a. A neuron applies (activates) the activation function to the sum of its inputs and adds its bias term to generate its output, i.e. $y = f_a(\sum_i \mathbf{x}_i) + Bias$. Activation functions used in the paper include the rectified linear unit, $ReLU(x) = max(0, x)$, and linear, $LINEAR(x) = c \cdot x$ where c is some constant.

Neural networks are composed of layers of neurons where signals are sent from one layer to another via connections. The first layer is the input layer, the last layer is the output layer, while internal layers are called as hidden layers.

Definition 12 (Artificial Neuron Connection) An artificial neuron connection $anc = (src^{(i)}, trg^{(j)}, w)$ leads from a source neuron $src^{(i)}$ (in layer i) to a target neuron $trg^{(j)}$ (in a subsequent layer j) with a given weight w. When neuron $src^{(i)}$ activates, the neuron connection multiplies the output of $src^{(i)}$ by its weight w and sets the *k*-th input element of $trg^{(j)}$ as the product ($trg^{(j)}.\mathbf{x}_k = w \cdot src^{(i)}.y$).

Definition 13 (Artificial Neural Network) An artificial neural network $ann = (\mathbf{x}, \hat{\mathbf{y}}, AN, ANC)$ contains a vector of inputs $\mathbf{x}$, a vector of outputs $\hat{\mathbf{y}}$, a set of artificial neurons AN arranged in n layers (where each neuron $an^{(i)} \in AN$ belongs to exactly one layer $1 \leq i \leq n$), and a set of artificial neuron connections ANC. The input $\mathbf{x}$ is composed of (the single) input of neurons in layer 1, i.e. $\mathbf{x}_i = an_i^{(1)}.x$ while the output $\hat{\mathbf{y}}$ is composed of (the single) output of neurons in layer n, i.e. $\mathbf{y}_j = an_j^{(n)}.y$. The output is calculated by activating each neuron in each layer sequentially.

Definition 14 (Neural Network ML Model) A neural network ML model $NN = (\mathbf{x}, \mathbf{y}, Data, Arch, HP)$ is an ML model in which the $Arch$ is defined as an artificial neural network ann. Hyperparameters HP include the number of neuron layers n, the number of neurons per each layer, and the activation functions used within the architecture.

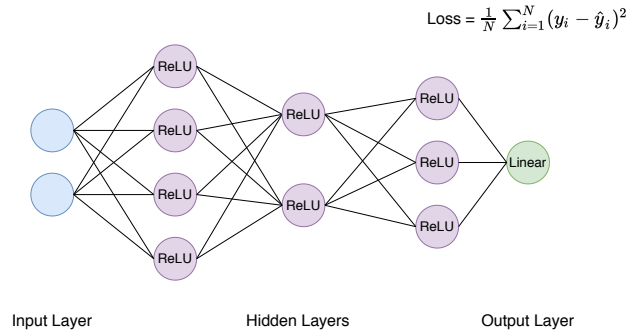


Fig. 7 Neural Network Architecture Setup.

Neural networks learn from data by adjusting the weights of their connections. Such adjustments are performed by an optimization algorithm (optimizer) aiming to minimize a given loss function. Many such optimizers are derivative-based, which impose further assumptions on the loss and activation functions.

7.3 Configuration and Training of ML Architectures

7.3.1 Bayesian Ridge Regression

We use the implementation of Bayesian ridge regression in the Scikit-Learn library [1]. This implementation uses four hyperparameters $(\alpha_1, \alpha_2, \lambda_1, \lambda_2)$ to provide Gamma distribution priors over the α and λ parameters. We set $\alpha_1 = \alpha_2 = \lambda_1 = \lambda_2 = 10^{-6}$ to maintain non-informative priors at the start. Training occurs by fitting to the training data set.

7.3.2 Neural Networks

Architecture setup We use the TensorFlow framework [3] to develop NNs. A similar architecture (Figure 7) is used for all cases: all neurons within hidden layers use the ReLU activation function, and output neurons have linear activation functions. ReLU enables non-linear learning and the linear activation allows for a wide range of bearing load predictions.

Each NN uses the mean square error loss function and the Adam optimizer [20] to optimize connection weights. Mean square error was chosen because it is differentiable and penalizes large outliers. The Adam optimizer is a standard optimizer used in ML; it has been found to be an effective optimization algorithm for most problems.

Training Each neural network is trained for 1000 epochs (with data shuffled at the end of each epoch and a batch size of 32) over the training data set. Early stoppage is not used as no overfitting was experienced [33].

Tuning hyperparameters Given the computational constraint L imposed to obtain deployable ML models (see Section 6.1), we classify neural networks into two complexity categories. *Simple* neural networks are allowed to perform at most *L/2* computational operations while *complex* neural networks are given a constraint of L computational operations.

While enforcing these computational constraints on the number of operations, we tune the number of neurons, the number of neurons per layer, and the number of neuron layers as hyperparameters. The best performing combination of hyperparameters for each simple and complex NN (as measured on the validation set) is then evaluated on the test set.

7.4 Data and Feature Engineering

Our process includes generating data, distilling the data into a dataset, feature engineering, and splitting data into training, validation, and test sets.

7.4.1 Data Generation

Data is generated from real engineering model simulations. The appropriate engineering models for each performance, aerothermal, and secondary air system are provided by engineers from each discipline.

A tool runner (script) launches engineering performance, aerothermal, and secondary air system model simulations sequentially. The engine operating conditions are defined for each performance run (engine power, altitude, ambient temperature, etc.). The performance outputs are then fed as inputs to the secondary air system model simulator with aerothermal parameters.

The tool runner runs simulations across a vast number of operating conditions to generate more than 150,000 engineering data points (while traditional practice relies on less than 200 points). The simulation results are organized and saved into a storage location that can be queried.

7.4.2 Data Distillation

The data distiller links data across each of the engineering discipline simulations into a common dataset for training ML models. The distillation process extracts on-engine sensor values and internal-engine parameters from performance simulations, and parameters necessary to calculate bearing loads from the SAS simulations. If a simulation run terminates with an error, (e.g. convergence failure) then this run is excluded from the distilled data set.

7.4.3 Feature Engineering

Due to the rigorous requirements of our engineering context, we have clearly defined inputs (sensors) and outputs.

As such, real feature engineering is limited to determining bearing load values from SAS simulation data. This involves a sequence of arithmetic calculations on a number of distilled SAS parameters.

7.4.4 Data Sets

We split the data set into training, validation, and test sets. We randomly select 10% of data for training, 10% for validation, and 80% for test.

We deviate from common ML practice of selecting a high percentage of data for training and a much smaller percentage for test due to the safety-critical nature of the prediction problem. We must ascertain that our prediction module generalizes well to unseen cases. In our engineering context, our prediction module needs to learn an underlying physics interaction or equation. As the module is learning to approximate solutions to a series of physics equations (only changes are variable values), an excessive number of data points should not be required. Given that a large amount of data is generated (150,000+ points), 10% should be sufficient to train effective predictors.

To experimentally justify the 10/10/80 data split, Figure 8 presents RMSE results for 1ML architectures, i.e. for a single (individual) prediction component. There are minimal differences between training, validation, and test set scores, and only for one case is there a larger RSME result on the test set than encountered in training or validation. MO and MU values metrics increase in the test set as compared to training, but with a larger sample of data more outliers are expected. The increases in MO and MU values are within a reasonable range. While we limit our presentation here to only 1ML architectures for simplicity, all ML models tested exhibit minimal differences in RMSE between data sets.

8 Experimental Evaluation

In order to evaluate RQ1 and RQ2,this section presents results for each of the 1ML, HSML, HPML, 2ML architectures presented in Section 7.1 evaluated on MO, MU, MAE, and RMSE metrics as described in Section 6.2. Evaluations are conducted on the *test set* described in Section 7.4.4 consisting of 120,000 data points for each architecture. Moreover, each architecture implementation is compared with the existing traditional prediction module as a baseline.

Results (normalized) are presented for three bearings: Bearing 1, Bearing 2, and Bearing 3. Extra details are provided for Bearing 1, and an overview for all three. These results represent predictions in real Siemens engines; precise details are masked and normalized as they represent business-sensitive data.

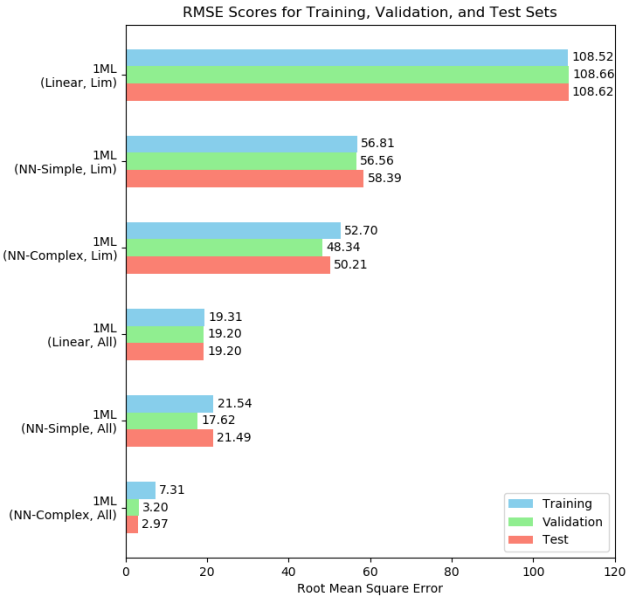


Fig. 8 RMSE metric results for each train, validation, test split. Evaluated for 1ML architectures.

8.1 1ML: Single ML Model

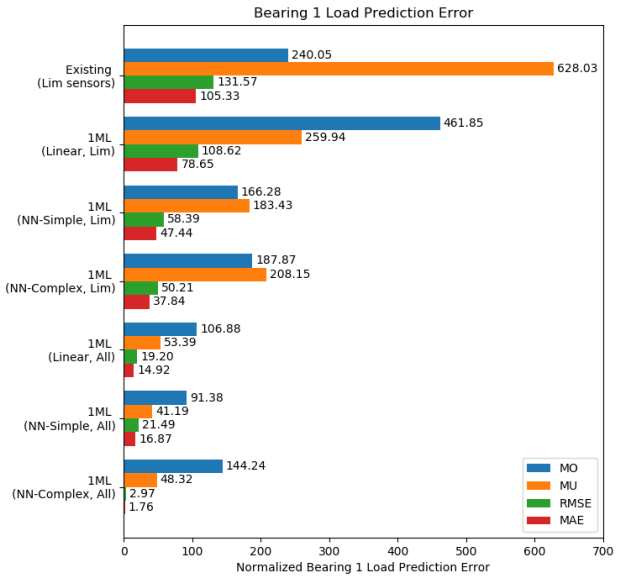
First, we create ML models which predict Bearing 1 load from the on-engine sensors (both **lim** and **all**) as a single AI component. We explore the performance of each ML model architecture and effect of the different inputs sets of sensors.

We compare six prediction modules with **lim** and **all** sensors for each of the linear, neural network (simple), and neural network (complex) architectures described in Section 7.1). Figure 9 presents the evaluated results.

Observation 1 suggests that ML-augmented prediction modules can improve on traditional techniques, but they may not always perform better for every metric. Observation 2 highlights that certain ML models can exhibit poor performance, but this was not a common case. Observation 3 reveals that the ML module benefits from access to more sensor data. Observation 4 is most easily observable when comparing neural network (simple) with neural network (complex). While neural network (complex) performed better with respect to MAE (Improvement of up to 60x) and RMSE, it performed worse in terms of MO and MU. This is important to note when deciding on a *deployment model* to deploy to *production*, i.e. how to balance worst case performance (MO, MU) with average performance.

8.2 HSML: Hybrid Model with ML SAS Component

In this section we present results for the HSML modules. We begin by evaluating the SAS components individually with



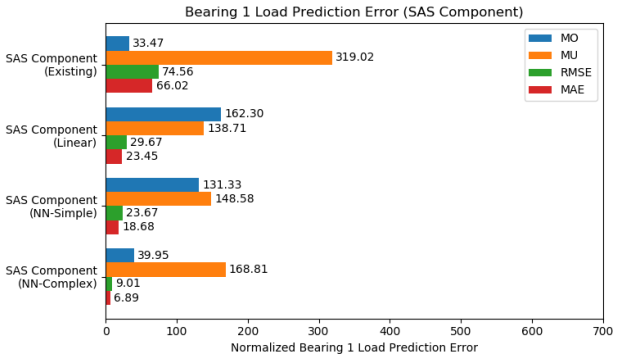
Observation 1: All 1ML modules outperformed the existing module wrt. MU, MAE, and RMSE.

Observation 2: Only Linear (lim) 1ML module exhibited worse MO than existing module.

Observation 3: Each 1ML (all) module outperformed all 1ML (lim) and existing module in all metrics.

Observation 4: More complex ML models always achieved better MAE and RMSE, but not always better MO and MU. 1ML(NN-Complex, All) decreased MAE by almost 60x.

Fig. 9 1ML prediction modules for Bearing 1.



Observation 1: The existing traditional SAS component achieves better performance in all metrics than the complete existing module.

Observation 2: Each ML-augmented SAS component outperformed the existing traditional SAS component wrt. MU, MAE, and RMSE.

Observation 3: Each ML-augmented SAS component had much worse MO than the existing SAS component.

Observation 4: The more complex ML models achieved better MAE and RMSE scores.

Observation 5: Each ML model architecture (linear, neural network) achieved worse MAE and RMSE as a SAS component than as a 1ML (all sensors) architecture in Figure 9.

Fig. 10 Prediction by SAS component (tested individually) for Bearing 1.

respect to the existing traditional SAS component and then present the evaluation of the entire hybrid module.

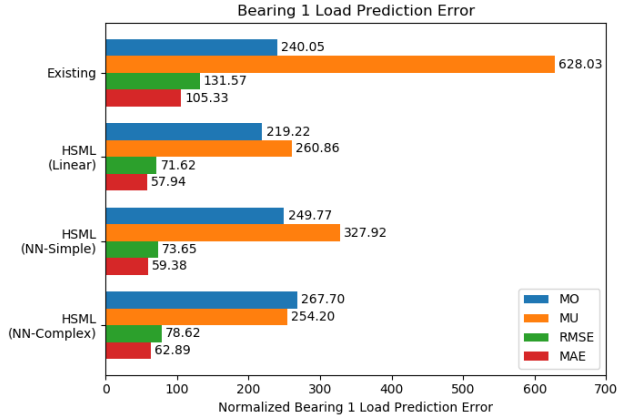
Figure 10 presents results for SAS components evaluated independently (not full prediction module). When analyzing these results, there are several key observations.

Observation 1 suggests there exists a propagation of error across components in the existing traditional module. Errors in the performance component further propagate with inaccurate predictions in the SAS component. Observation 2 presents further evidence that ML-augmented components can improve upon existing techniques. Observation 3 shows that components can be biased towards a specific metric - clearly the existing traditional SAS component is biased towards underprediction. Observation 4 shows that even more complex models (beyond computation limit imposed) could yield even better performance metrics. Observation 5 suggests that all on-engine sensors could provide more relevant information for predicting the bearing load than the predefined SAS component internal engine parameter inputs.

We have several ML model components which could serve as replacements for the existing SAS component. Each of these ML model components perform better in regards to MAE and RMSE. We now replace the existing traditional SAS component for each of these new ML model components in the existing module.

The existing performance module is kept unchanged and the existing SAS component is swapped for each trained ML model component in Figure 10. Thus, the traditional performance component predicts the internal parameters from the limited sensor input and a SAS component predicts the Bearing 1 load. Figure 11 presents results.

Observations 1 and 2 show that replacing existing components with ML-augmented components can improve module performance, but improvement may be limited by other components. Observation 3 is particularly important as it is a negative result. We observed that *improvement in a single component in a system could actually decrease the overall performance of the system*, thus violating compositionality for predictor components. It is therefore paramount to re-execute integration and system-level testing to investigate the system behaviour in its entirety before deploying "improved" components.



Observation 1: Replacing the existing traditional SAS component with ML model components improved the prediction module wrt. MAE, RMSE, and MU.

Observation 2: MO performance did not significantly change between any of the SAS component implementations.

Observation 3: While neural network SAS components achieved better MAE and RMSE measures independently as compared to the linear model component (Figure 10), the full prediction integrating neural network SAS components performs worse than the one integrating the linear model SAS component.

Fig. 11 HSML prediction modules for Bearing 1.

8.3 HPML: Hybrid Model with ML Performance Component

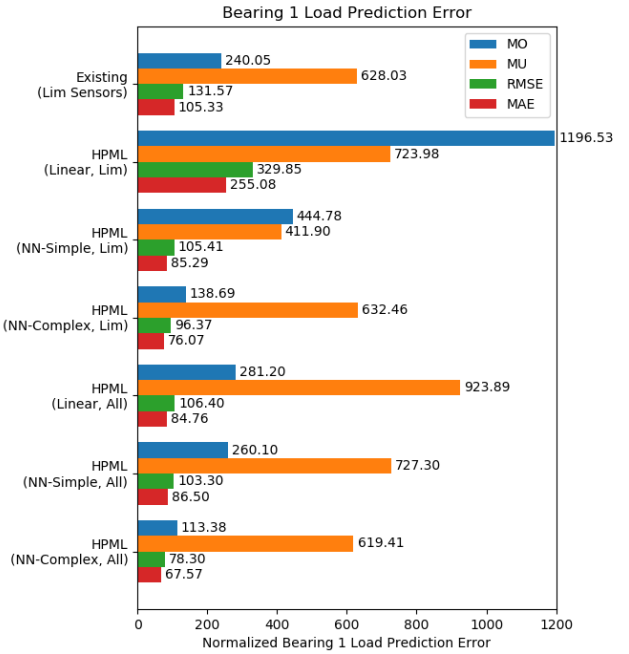
We present the results of a prediction module with the HPML architecture. We do not show independent ML performance component evaluation as it is difficult to present concisely (multi-parameter) and provides the reader little value. Figure 12 presents the results of HPML architectures.

Observations 1, 3, and 4, highlight that ML-augmented components cannot guarantee improved performance in all defined metrics. Observation 2 supports the hypothesis of compounding propagated error between components. With an improved performance component, the overall module MAE and RMSE approached the MAE and RMSE of the existing SAS component individually.

8.4 2ML: Dual ML Model

In Section 8.2 and Section 8.3 we developed a number of ML-augmented SAS and performance components and evaluated hybrid models mixing existing components with the new ML-augmented components. In this section we study the 2ML architecture, by connecting two ML model components to form the prediction module.

We present a subset of the available permutations which we believe provides the most value to the reader (more permutation results presented in Section 8.5. In this section,



Observation 1: Only the HPML(NN-Complex, All) module outperformed the existing traditional module in all performance metrics.

Observation 2: The neural network (complex, all) performance component model brought the MAE (67.57) and RMSE (78.30) of the entire system near the MAE (66.02) and RMSE (74.56) of the independently evaluated existing SAS component.

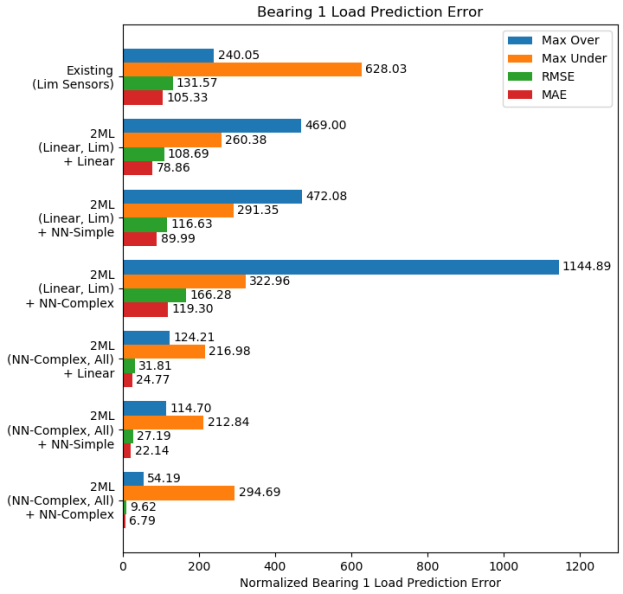
Observation 3: All ML model performance components except for the Linear (Lim.) improved full module MAE and RMSE.

Observation 4: No ML model performance component provided significant module improvements for both MO and MU.

Fig. 12 HPML prediction modules for Bearing 1.

we compare all permutations arising from Linear (Lim.) and NN (All, Complex) for the performance component and all three of Linear, NN (Simple), and NN (Complex) for the SAS component. Each of these are presented in relation to the existing module in Figure 13.

From Observations 1 and 2 we show that different ML model components have different sensitivities to input parameter noise. There clearly exist strong interaction effects when component models are replaced. Unfortunately, these effects are not easy to predict. Components must be evaluated as integration to a whole. Observation 3 presents that with accurate (MAE, RMSE) prediction components, the propagation of error is quite low.



Observation 1: Using the Linear (Lim.) model as performance component, the increasing complexity of the SAS component model decreased the overall performance in all four metrics.

Observation 2: When the performance component is set to NN (All, Complex), the more complex components achieve better MAE and RMSE values.

Observation 3: Using the NN (All, Complex) model as performance component, each module achieved almost the same RMSE performance as the SAS performance component did individually.

- Linear: 29.67 (indep. SAS) vs. 31.80 (2ML)
- NN (Simple): 23.67 (indep. SAS) vs. 27.19 (2ML)
- NN (Complex): 9.01 (indep. SAS) vs. 9.61 (2ML)

Fig. 13 2ML prediction modules for Bearing 1.

8.5 Summary

In the previous subsections, we presented results for Bearing 1 prediction modules incorporating the 1ML, HSML, HPML, and 2ML architectures. Now we summarize our results for each architecture for Bearing 1, Bearing 2, and Bearing 3. We omit results pertaining to limited sensor input (except for the existing traditional module) as modules limited to these sensors achieve much worse performance.

Results for Bearings 1, 2, and 3 are presented in tournament style tables in Figure 14. Pairings were done in sequential order of Existing, 1ML, HSML, HPML, and 2ML module architectures. Note: certain modules may have advanced further (or even won) in the tournament if paired with different modules - important observations can be found by comparing non-competing modules. To help the reader navigate through the table, only the better metric result for each "competition" is coloured.

Observations 1 and 3 suggest that ML modules can be effective replacements for existing prediction modules. Best performing 1ML modules (in terms of MAE) achieved significant 60x, 22x, and 17x MAE reduction for Bearings 1, 2, and 3, respectively. Observation 2 suggests that integrated one-component modules may be a recommended option for prediction modules to decrease error propagation as well as computation costs. As a final remark, no ML model trained for each prediction task suffered from overfitting.

RQ1: How effective is it to replace an individual (traditional) prediction component or a chain of components (module) with ML-driven counterparts?

- Experimental results for predicting bearing loads in a gas turbine engine suggest that ML-driven prediction components can be more effective than traditional prediction components.
- Results also show the errors of individual components may propagate along a chain of prediction components thus degrading end-to-end prediction performance. As such, *the principle of compositionality is violated.*
- As a consequence, special attention is needed for the replacement of individual components within a chain as an "improved" component may degrade the overall prediction of an entire module (component chain).
- In general, replacing components earlier in a prediction chain is recommended (though no guarantees are provided for the best overall result).

RQ2: How effective are different combinations of ML prediction components used for engineering models of gas turbines?

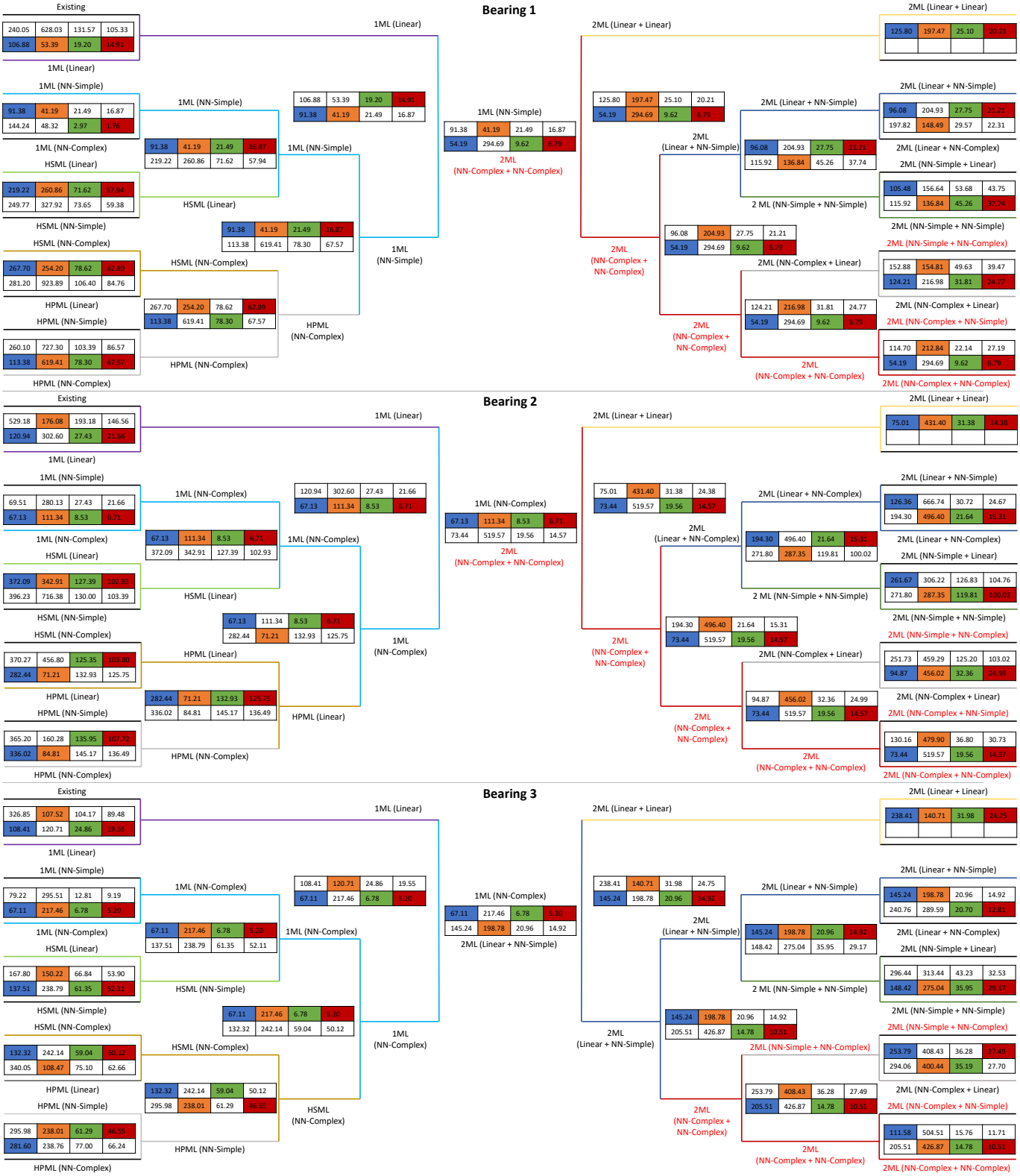
- Replacing multiple components with a combination of ML prediction components is demonstrated to improve upon existing traditional prediction modules in our gas turbine case study, However, results suggest that replacing chains (modules) of prediction components with a single ML component may be recommended.

9 Deployment

Next, we present an automated technique for creating deployment models from validated models (with a focus on neural networks due to their complexity) and deploying them into production in the engine control system running on a programmable logic controller (PLC) hardware. We propose to use code generation techniques that treat ML models as simple deployment model artifacts.

9.1 Hardware

The hardware on which the control system of such engines is running is a Programmable Logic Controller (PLC). PLCs



Observation 1: 1ML and 2ML architectures routinely outperform the existing, HSML, and HPML architectures.

Observation 2: 2ML architecture composed of two NN-complex components only outperformed 1ML architectures in Bearing 2 (note that 1ML (NN-Complex) only exhibits inferior MO in comparison), despite having more than twice the allotted computation (does not fulfill deployment requirement).

Observation 3: ML-based prediction modules were able to outperform the existing traditional prediction modules in all three bearings on all metrics except for MU in bearing 3.

Fig. 14 Tournaments are presented where module architectures are paired against each other to visually filter a "best performing" module architecture for each bearing. Tournament advances are decided by superior performance (colour indicates superior, white inferior) on a majority of metrics (tie-breaker is MO - most dangerous scenario). Metrics in each match follow the same order and colouring schema as in previous figures: MO (blue), MU (orange), RMSE (green), MAE (red). Models with red names fail to meet the computation requirement; nevertheless, we leave them as they may help the reader better understand the capability of each architecture.

are primarily designed to be robust, capable of functioning in any environment, and consistent. As such, these controllers are single-threaded and do not support dynamic memory allocation or code optimization.

Real-time programs loaded into a PLC controller are looped continuously in time intervals until the controller is turned off. Each program is given a time frame (e.g. 10ms) within which it must complete. It is common practice to assign different priorities and allowed time to different real-time programs. Programs with greater priority may preempt or interrupt lower priority programs, execute, and then relinquish control. This is illustrated in Figure 15.

Fig. 15 PLC time interval logic. 1 has highest priority, 3 has lowest. 1 has 10ms time interval and requires 5ms, 2 has 30ms time interval and requires 10ms, 3 has 60ms time interval and requires 2.5ms.

Given that the controller has a slow processor and the control system has strict, hard real-time requirements, any deployment model must be computationally efficient to be able to complete its prediction in time. Deployment to PLCs is currently limited to *nonadaptive models*.

9.2 ML Model Format

A feed-forward neural network model is simply a mathematical function, composed of smaller mathematical operations occurring in layers. As such, it is possible to encode, or represent, the model as a sequence of operations in a standardized format. This is the basis for how Tensorflow, Keras, Pytorch, MATLAB, etc. allow one to save a model and reload it later. However, each contains its own standard, and for the purposes of a PLC, encapsulates extraneous information, such as training-time parameters and behaviour. Thus, we developed a new model representation for storing *validated* models for PLC deployment to prevent vendor lock-in.

Our neural network ML model uses the JavaScript Object Notation (JSON) format. The JSON file maintains a list of layers (feed forward neural networks can only receive inputs from previous layers). For each layer in the model, the following attributes are maintained:

- **Name:** The given name for a layer will be used to refer to layer computations in the generated PLC code.
- **Activation function:** The generated PLC code will call appropriate activation function instructions, which is assumed to be identical within a layer. If multiple activation functions are desired in a layer, the layer can be split into two parallel layers.
- **Size:** Number of neurons in the layer.
- **Input size:** Number of inputs for each neuron.
- **Input layers:** List of references to layers which serve as input to the layer.
- **Weights:** Weights for each connection from neurons in the input layers to the neurons in the layer.
- **Bias:** Biases for each neuron in the layer.

Each ML model is tagged with a model type, version number, author, date, training set data id, test set id, and model metric scores. JSON documents are easily comparable to find differences between models. Listing 1 showcases an example JSON file.

```
"model_type": "Feed-Forward Neural Network",
"version": "Bearing1_A.1.1",
"date": "2020-01-01T01:00:00+00:00",
"training_id": "Bearing1_A_trn_2020",
"test_id": "Bearing1_A_tst_2020",
"metrics": {
  "MO": 93.13,
  "MU": 109.41,
  "RMSE": 51.23,
  "MAE": 39.32,
},

"layers": [
  {INPUT LAYER},

  {
    "NAME": "HIDDEN",
    "ACTIVATION_FUNCTION": "RELU",
    "SIZE": 3,
    "INPUT_SIZE": 1,
    "INPUT_LAYERS": ["INPUT"],
    "WEIGHTS": [[0.031, 0.041, 0.058]],
    "BIAS": [0.003, 0.024, 0.009]
  },

  {OUTPUT LAYER}
],

"layers_ptr": {
  "INPUT": 0,
  "HIDDEN": 1,
  "OUTPUT: 2
}
```

Listing 1 JSON neural network example. Input and output layers are omitted to simplify presentation.

9.3 Source Code Generation

To deploy the JSON encoded neural network onto a PLC, we use code generation techniques. Given that we have a unified encoding of a sequence of mathematical operations, we need to correctly unfold the sequence, and apply the proper operations at the right time.

PLC code can be composed of routines and functions. Routines define an execution order of instructions and allow for jumps to other routines. Upon completion of a routine which was jumped to, the caller routine continues execution. Functions are reusable pieces of code which can be called in a routine with provided variables.

The prediction model deployed to the PLC uses the following generated logic. As feed-forward neural network can

only receive input from previous layers, the activation of layers occurs sequentially where each layer has its own routine. Each neuron activates in a layer, and then the next layer's routine is run. This process is repeated until each defined layer in the model is generated.

A sample "main" routine which executes the core logic of a prediction program is presented in Listing 2.

```
// Normalization of inputs
Execute routine APPLY_INPUT_NORM;

// Populates inputs to neurons in HIDDEN layer
Execute routine APPLY_WEIGHTS_BIAS_HIDDEN;

// Each HIDDEN neuron activates and stores output
Execute routine APPLY_ACTIVATION_HIDDEN;

// Populates inputs to neurons in OUTPUT layer
Execute routine APPLY_WEIGHTS_BIAS_OUTPUT;

// Each OUTPUT neuron activates and stores output
Execute routine APPLY_ACTIVATION_OUTPUT;

// De-normalization of OUTPUTS
Execute routine APPLY_OUTPUT_DENORM;
```

Listing 2 "Main routine" example.

9.3.1 Activation Functions

Activation functions are generally simple mathematical operations that take one input and return one output. Each neuron in a neural network requires an activation function. The code generator contains templates for standard, named activation functions (ReLU, Linear, etc.). As usual, templates for each PLC programming language are derived from previously developed code.

Definition 15 (PLC activation function) A generated PLC activation function $PAF = (In, Out, fn, Name)$ defines the input memory location In, the output location Out, the function code fn applied to the input to get the output, and the activation function name $Name$.

We use a naming convention to name each activation function as NN_ActivationFunctionName (e.g. NN_Linear) for consistency and to create a library of (template) functions for the PLC.

The code generator extracts all types of activation functions used within the NN from the validated ML model and then extracts the proper activation function PLC code from its template library. Each activation function is generated as its own independent function and will be used as such. When a neuron "activates", it calls the relevant function from the defined function sets. An example of what a RELU activation function template would like is presented in Listing 3.

```
Function NN_RELU(INPUT, OUTPUT) {
    If   (INPUT > 0.0) OUTPUT = INPUT;
    Else               OUTPUT = 0.0;
}
```

Listing 3 Pseudocode RELU activation function example.

9.4 Layer Generation

Each neuron layer has two generated routines from the validated ML model described in Section 9.2. Each layer contains references to defined arrays with naming convention OUTPUT_LayerName. Each array element serves as a storage location for neuron - it captures neuron input value and provides the output value post-activation.

Determining the output for each neuron occurs in two stages: (1) first, we compute the input (routine referred to as APPLY_WEIGHTS_BIAS_LayerName) and then (2) we apply the activation function to get the output (routine referred to as APPLY_ACTIVATION_LayerName). The input is determined by adding the neuron's bias term with the sum of products between weights and their corresponding neurons outputs from previous layers (referenced by their corresponding OUTPUT_PrevLayer names, see Listing 4. This input is stored in the array, after which the activation function is applied (Listing 5) and overwrites the array value.

```
OUTPUT_HIDDEN[0] =  OUTPUT_INPUT[0] * WEIGHTS_HIDDEN[0][0]
                      + BIAS_HIDDEN[0];
OUTPUT_HIDDEN[1] =  OUTPUT_INPUT[0] * WEIGHTS_HIDDEN[0][1]
                      + BIAS_HIDDEN[1];
OUTPUT_HIDDEN[2] =  OUTPUT_INPUT[0] * WEIGHTS_HIDDEN[0][2]
                      + BIAS_HIDDEN[2];
```

Listing 4 Pseudocode APPLY_WEIGHTS_BIAS_HIDDEN example.

```
Execute func NN_RELU(OUTPUT_HIDDEN[0],OUTPUT_HIDDEN[0]);
Execute func NN_RELU(OUTPUT_HIDDEN[1],OUTPUT_HIDDEN[1]);
Execute func NN_RELU(OUTPUT_HIDDEN[2],OUTPUT_HIDDEN[2]);
```

Listing 5 Pseudocode APPLY_ACTIVATION_HIDDEN example.

9.5 Configuration Generation

PLC code must be packaged in a way that enables deployment into the control system. We define a configuration file which contains named routines with auto-generated source code, initializes all data structures, and metadata.

Routines Routines are composed of generated source code. For the purpose of consistency for code generation, each routine follows a defined naming convention.

- APPLY_WEIGHTS_BIAS_LayerName for routines that compute neuron inputs.
- APPLY_NORMALIZATION_LayerName for routines that apply normalization.
- APPLY_ACTIVATION_LayerName for routines which apply the activation function for each neuron in a layer.

Data structures Each layer in a NN needs three defined data structures: an array for weights, an array for biases, and an array for storing layer neuron outputs. Each array is composed of floating point numbers. For optimization purposes, the weight and bias arrays are defined as constants.

Metadata The generated configuration file defines several key pieces of data such as the author, the generation date, and the program name.

9.6 Code Optimization

In the context of the control system, the code of the deployed predictors need to run at real-time, thus the efficiency of the code is important. However, via a series of experiments, we determined that not all PLCs have effective compilers. Optimizing changes in the code syntax (oftentimes to the detriment of readability) could have profound changes on the execution time of code.

One such optimization revealed via experimentation was to avoid separating variable operations into multiple lines. Due to a lack of compilation, it appears the PLC performs extra unnecessary read and write operations. One can achieve 25%+ faster execution times by only setting a variable ($var =$ ...) once and keeping all operations on one line of code, albeit, the line may become 1000+ characters long if a layer is composed of many neurons.

```
OUTPUT_HIDDEN[0] = OUTPUT_INPUT[0] * WEIGHTS_HIDDEN[0][0]
                   + ... + ... + BIAS_HIDDEN[0];
```

Listing 6 Pseudocode PLC optimization example.

Obviously, code performance improvements are more important than readability for such a prediction module, especially, since the code is auto-generated and it should not be touched by control systems engineers.

9.7 Physical Deployment

Unfortunately, physical deployment, cannot be fully automated in the context of gas turbines. The process of physically deploying the prediction module to the control system involves control system engineers importing the generated code and configurations, properly connecting the on-engine sensors to the module, and uploading the control system code onto the physical PLC.

9.8 Deployment constraints for ML architectures

A real-time program must guarantee to complete within its allocated execution time. The available time is imposed by the PLC hardware, which may define restrictions and constraints for the underlying ML architectures. Even if a given ML architecture performs particularly well during training and validation, it cannot be used for runtime predictions if the architecture cannot be deployed to the PLC. As such, our goal was to identify some deployment constraints for ML modules that can be enforced at design-time to enable their deployment to the production environment.

Through a series of experimental tests on physical PLC hardware, we are able to determine how much time certain floating point, integer, read and write operations require. This knowledge can help us place constraints on neural network training model size (Section 6.1).

The majority of NN computation is spent on the multiplication of neuron output values by their respective weights. Thus *the total computation time of a NN model is dominated by the number of existing connections*. Using our experimentally determined hardware computation times, we can put a conservative constraint on the number of connections in a neural network training model to ensure that the deployed module will satisfy timeliness related requirements.

9.9 Overview

In this section, we used code generation to automate the creation of optimized ML deployment models for a specific production hardware to address RQ3.

RQ3: How to automate the deployment of a trained ML model to a CPS hardware platform to improve maintainability?

– Automation of trained ML model deployment can be accomplished via code generation techniques to develop source and configuration files required for a PLC hardware platform. ML models and relevant metadata can be represented within template artifacts. Given code generation and configuration scripts, complex ML models can be maintained (upgraded, replaced, etc.) without significant engineer involvement.

10 Threats to Validity

Construct Validity To limit threats to construct validity from a design perspective, our test metrics directly relate to inputs and outputs of the existing prediction module and the correctness of simulation data was validated by Siemens engineers. While we exclusively rely upon simulation data (instead of real field data) for training, this is unavoidable in our engineering context as bearing loads cannot be measured directly. In addition, the output of our predictors has been validated by using independent system-level runtime simulators regularly used in engine design. During those tests, no warnings or errors were reported.

Internal Validity We mitigate threats to internal validity by using implementations of ML techniques from trusted and popular ML libraries (Scikit-Learn and TensorFlow) and following machine learning best practices. Likewise, we compare predicted results to actual design time results and validate on 120k unique simulation runs in the test set for each

of three bearings of a real gas turbine engine. For validation, each of these three bearings underwent black-box and system-level testing. Additionally, to increase the level of confidence in predictions, we provide metrics evaluating worst case over and under-prediction in addition to standard error metrics used in ML.

External Validity While we carried out extensive evaluation of ML-based predictors in the context of gas turbine design, we do not claim that similar results and findings would be obtained for other CPSs. In particular,

- The data sets within this paper arise from specialized non-chaotic physics simulations. For other data sets, other ML techniques may be more effective or may not perform better than existing traditional predictors.
- Our deployment optimizations are specific to PLC hardware and the compilers which were used, and thus may not generalize to other hardware platforms.
- While in our context, it is frequently more effective to replace earlier prediction components in a chain, this may not always be the case (e.g. information loss such as lossy channel between).

On the other hand, our key negative finding that the principle of compositionality can be violated within chains of prediction components should generalize (by definition). We do not see any inherently special characteristics of our system which would limit this system-level finding to the context of designing prediction components for gas turbines. As such, developing ML-based predictors which exhibit compositional behavior is a major open challenge.

11 Conclusions

In this paper, we addressed the problem of applying and deploying machine learning predictors to gas turbine design from a systems engineering perspective. Given the safety-critical nature of gas turbines and the interdependence of existing subsystems developed by coordinated efforts of many multidisciplinary engineering teams working on individual components and modules, architectural changes in the system are difficult and rare.

For this purpose, we proposed and evaluated four architectures (1ML, HSML, HPML, 2ML) for potentially replacing existing bearing load prediction modules with ML-driven counterparts. Despite using only 10% of data for training, for each of these architectures and for both Bayesian ridge regression and NNs, the models generalized and exhibited very minimal differences in performance between training, validation, and test sets.

Additionally, we showcased how an ML model can be automatically deployed and integrated into an *off-the-shelf* PLC hardware platform. We provided various insights into deployment as well as source code optimization. For example, how to determine and incorporate platform and computation requirements for training ML models to ascertain that the deployed ML predictors can actually run on the designated platform.

Conceptual Contributions

- We demonstrated the efficacy of applying ML for prediction purposes within a gas turbine CPS control system. With ML, we managed to reduce mean absolute error (vs traditional methods) by up to 60x and decrease worst-case over and under-predictions.
- We evaluated the effects of replacing existing system components with ML-driven counterparts. In some cases, prediction module performance dropped even if a component was replaced with a seemingly better (when evaluated as an individual black box) ML-driven counterpart. Integration testing is key, as interaction between components is difficult to predict. Thus, as a key negative finding, we experienced the violation of compositionality in components of prediction chains, thus providing a major barrier for incremental re-certification.
- We proposed and validated methodology for automating deployment of ML models into a low-level control system of gas turbines. By using code generation from a well-defined ML model template, we were able to deploy neural networks and linear models onto a PLC.

Engineering/Industrial Impact

- Thanks to the automated deployment of prediction modules, over 20 engineering workdays are saved each time an update to a prediction module is required.
- Within our deployment framework we incorporated versioning to distinguish between prediction modules. This improved version comparisons and the ability to revert to previous versions easily if necessary.
- By using code generation, we showed how a prediction module can be deployed to multiple supported PLC platforms from one ML model artifact. This reduces the number of software defects thus increasing software quality across multiple supported hardware platforms.

The work presented in this paper will be continued at Siemens Energy. It will be applied to other subsystems and deployed in the field in new and revised engines.
Further studies, especially in other CPS domains, would help validate opportunities of automated code generation for deployment of ML in existing systems. This would provide other unique case studies with different data sets and hardware platforms which would greatly decrease existing threats to external validity. We believe researching methods which improve compositionality for prediction components would be highly beneficial to the industry at large.

Acknowledgements This work was partially supported by the Digital Multidisciplinary Analysis and Design Optimization Platform for Aeroderivative GasTurbines (Siemens Ca CRDPJ 513922-17 X-247371 and NSERC CRDPJ 513922-17 X-247323 funds)

References

1. Scikit-Learn Bayesian Ridge Regression. URL https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.BayesianRidge.html#sklearn.linear_model.BayesianRidge
2. SGT-A65: Aeroderivative gas turbine: Gas turbines: Manufacturer: Siemens energy global. URL https://www.siemens-energy.com/global/en/offerings/power-generation/gas-turbines/sgt-a65-tr.html
3. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015). URL https://www.tensorflow.org/. Software available from tensorflow.org
4. Alexopoulos, K., Nikolakis, N., Chryssolouris, G.: Digital twin-driven supervised machine learning for the development of artificial intelligence applications in manufacturing. International Journal of Computer Integrated Manufacturing **33**(5), 429–439 (2020). DOI 10.1080/0951192X.2020.1747642
5. Bencomo, N., Paucar, L.H.G.: RaM: Causally-connected and requirements-aware runtime models using bayesian learning. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 216–226. IEEE (2019)
6. Bishop, C.M.: Pattern recognition and machine learning. Springer (2006)
7. Boschert, S., Rosen, R.: Digital Twin—The Simulation Aspect, pp. 59–74. Springer International Publishing, Cham (2016). DOI 10.1007/978-3-319-32156-1_5
8. Breuker, D.: Towards model-driven engineering for big data analytics–an exploratory analysis of domain-specific languages for machine learning. In: 2014 47th Hawaii International Conference on System Sciences, pp. 758–767. IEEE (2014)
9. Burgueño, L., Cabot, J., Gérard, S.: An LSTM-based neural network architecture for model transformations. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 294–299. IEEE (2019)
10. Cengarle, M.V., Bensalem, S., McDermid, J., Passerone, R., Sangiovanni-Vincentelli, A., Törngren, M.: CyPhERS: Cyber-physical european roadmap & strategy characteristics , capabilities, potential applications of cyber-physical systems: a preliminary analysis. Tech. Rep. 611430 (2013)
11. Chbat, N.W., Rajamani, R., Ashley, T.A.: Estimating gas turbine internal cycle parameters using a neural network. In: ASME 1996 International Gas Turbine and Aeroengine Congress and Exhibition, pp. V005T15A023–V005T15A023. American Society of Mechanical Engineers (1996)
12. Darvas, D., Viñuela, E.B., Majzik, I.: PLC code generation based on a formal specification language. In: 2016 IEEE 14th International Conference on Industrial Informatics (INDIN), pp. 389–396. IEEE (2016)
13. Fan, C., Xiao, F., Zhao, Y.: A short-term building cooling load prediction method using deep learning algorithms. Applied energy **195**, 222–233 (2017)
14. Fast, M.: Artificial neural networks for gas turbine monitoring. Division of Thermal Power Engineering, Department of Energy Sciences ... (2010)
15. Gregory, B.: Turbine preliminary design using artificial intelligence and numerical optimization techniques. Journal of Turbomachinery **114**, 1 (1992)
16. Huyck, B., Ferreau, H.J., Diehl, M., De Brabanter, J., Van Impe, J.F., De Moor, B., Logist, F.: Towards online model predictive control on a programmable logic controller: Practical considerations. Mathematical Problems in Engineering **2012** (2012)
17. Ibrahem, I., Akhrif, O., Moustapha, H., Staniszewski, M.: Neural networks modelling of aero-derivative gas turbine engine: A comparison study. pp. 738–745 (2019). DOI 10.5220/0007928907380745
18. Kanelopoulos, K., Stamatis, A., Mathioudakis, K.: Incorporating neural networks into gas turbine performance diagnostics. In: ASME 1997 International Gas Turbine and Aeroengine Congress and Exhibition, pp. V004T15A011–V004T15A011. American Society of Mechanical Engineers (1997)
19. Kiakojoori, S., Khorasani, K.: Dynamic neural networks for gas turbine engine degradation prediction, health monitoring and prognosis. Neural Computing and Applications **27**(8), 2157–2192 (2016)
20. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
21. Kusiak, A., Li, M., Zhang, Z.: A data-driven approach for steam load prediction in buildings. Applied Energy **87**(3), 925–933 (2010)
22. Kusmenko, E., Nickels, S., Pavlitskaya, S., Rumpe, B., Timmermanns, T.: Modeling and training of neural processing systems. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 283–293. IEEE (2019)
23. Lafdani, E.K., Nia, A.M., Ahmadi, A.: Daily suspended sediment load prediction using artificial neural networks and support vector machines. Journal of Hydrology **478**, 50–62 (2013)
24. Lazzaretto, A., Toffolo, A.: Analytical and neural network models for gas turbine design and off-design simulation. International Journal of Applied Thermodynamics **4**(4), 173–182 (2001)
25. Lee, E.A., Hartmann, B., Kubiatowicz, J., Rosing, T.S., Wawrzynek, J., Wessel, D., Rabaey, J.M., Pister, K., Sangiovanni-Vincentelli, A.L., Seshia, S.A., Blaauw, D., Dutta, P., Fu, K., Guestrin, C., Taskar, B., Jafari, R., Jones, D.L., Kumar, V., Mangharam, R., Pappas, G.J., Murray, R.M., Rowe, A.: The swarm at the edge of the cloud. IEEE Design & Test **31**(3), 8–20 (2014). DOI 10.1109/MDAT.2014.2314600
26. Li, Q., Meng, Q., Cai, J., Yoshino, H., Mochida, A.: Applying support vector machine to predict hourly cooling load in the building. Applied Energy **86**(10), 2249–2256 (2009)
27. Luo, W., Hu, T., Zhang, C., Wei, Y.: Digital twin for CNC machine tool: modeling and using strategy. Journal of Ambient Intelligence and Humanized Computing (2018). DOI 10.1007/s12652-018-0946-5
28. Madni, A.M., Madni, C.C., Lucero, S.D.: Leveraging digital twin technology in model-based systems engineering. Systems **7**(1), 7 (2019)
29. Nascimento, R.G., Viana, F.A.: Fleet prognosis with physics-informed recurrent neural networks. arXiv preprint arXiv:1901.05512 (2019)
30. Nguyen, P.T., Di Rocco, J., Di Ruscio, D., Pierantonio, A., Iovino, L.: Automated classification of metamodel repositories: A machine learning approach. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 272–282. IEEE (2019)

31. Ogaji, S., Singh, R.: Artificial neural networks in fault diagnosis: A gas turbine scenario. In: Computational Intelligence in Fault Diagnosis, pp. 179–207. Springer (2006)
32. Pilarski, S., Staniszewski, M., Villeneuve, F., Varró, D.: On artificial intelligence for simulation and design space exploration in gas turbine design. In: L. Burgueño, A. Pretschner, S. Voss, M. Chaudron, J. Kienzle, M. Völter, S. Gérard, M. Zahedi, E. Bousse, A. Rensink, F. Polack, G. Engels, G. Kappel (eds.) 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019, pp. 170–174. IEEE (2019). DOI 10.1109/MODELS-C.2019.00029
33. Prechelt, L.: Early Stopping - But When?, pp. 55–69. Springer Berlin Heidelberg, Berlin, Heidelberg (1998). DOI 10.1007/3-540-49430-8_3
34. Puschel, M., Moura, J.M., Johnson, J.R., Padua, D., Veloso, M.M., Singer, B.W., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., et al.: SPIRAL: Code generation for dsp transforms. Proceedings of the IEEE **93**(2), 232–275 (2005)
35. Qiao, Q., Wang, J., Ye, L., Gao, R.X.: Digital twin for machining tool condition prediction. Procedia CIRP **81**, 1388 – 1393 (2019). DOI https://doi.org/10.1016/j.procir.2019.04.049. 52nd CIRP Conference on Manufacturing Systems (CMS), Ljubljana, Slovenia, June 12-14, 2019
36. Rauber, T.W., de Assis Boldt, F., Varejão, F.M.: Heterogeneous feature models and feature selection applied to bearing fault diagnosis. IEEE Transactions on Industrial Electronics **62**(1), 637–646 (2015)
37. Sacha, K.: Automatic code generation for PLC controllers. In: International Conference on Computer Safety, Reliability, and Security, pp. 303–316. Springer (2005)
38. Sobie, C., Freitas, C., Nicolai, M.: Simulation-driven machine learning: Bearing fault classification. Mechanical Systems and Signal Processing **99**, 403–419 (2018)
39. Steinegger, M., Zoitl, A.: Automated code generation for programmable logic controllers based on knowledge acquisition from engineering artifacts: Concept and case study. In: Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012), pp. 1–8. IEEE (2012)
40. Thapa, D., Park, C.M., Park, S.C., Wang, G.N.: Auto-generation of IEC standard PLC code using t-MPSG. International Journal of Control, Automation and Systems **7**(2), 165–174 (2009)
41. Thomas, G., Cabaret, S., Barillère, R., Kulman, N., Rochez, J., Pons, X., Azarov, K.: LHC-GCS: a model-driven approach for automatic PLC and SCADA code generation. Tech. rep. (2005)
42. Tipping, M.E.: Sparse bayesian learning and the relevance vector machine. Journal of machine learning research **1**(Jun), 211–244 (2001)
43. Wang, Z., Hong, T., Piette, M.A.: Building thermal load prediction through shallow machine learning and deep learning. Applied Energy **263**, 114683 (2020)
44. Widodo, A., Kim, E.Y., Son, J.D., Yang, B.S., Tan, A.C., Gu, D.S., Choi, B.K., Mathew, J.: Fault diagnosis of low speed bearing based on relevance vector machine and support vector machine. Expert systems with applications **36**(3), 7252–7261 (2009)