SMiLe – Automated End-to-end Sensing and Machine Learning Co-Design

Tanmay Goyal
ABB Research Center
Switzerland
tanmay.goyal@ch.abb.com

Pengcheng Huang
ABB Research Center
Switzerland
pengcheng.huang@ch.abb.com

Felix Sutton
ABB Research Center
Switzerland
felix.sutton@ch.abb.com

Balz Maag
ABB Research Center
Switzerland
balz.maag@ch.abb.com

Philipp Sommer
ABB Research Center
Switzerland
philipp.sommer@ch.abb.com

Abstract

Edge analytics is emerging as an important enabler for novel digital services. Appliances in smart homes and buildings, programmable logic controllers in automation processes or applications involving motors and pumps in powertrains are only a few amongst many examples that greatly benefit from analytics performed at the edge. The core principle of this new paradigm is to process data where it is generated. For instance, machine learning (ML) algorithms are directly running on typically small resource constrained embedded systems. In comparison to cloud analytics, edge analytics reduces data transmission burdens, improves data security and enhances the responsiveness of in-field devices when actions are needed based on local predictions.

The main challenge however is to fit an accurate processing pipeline, including both sensing and machine learning, into embedded systems. These systems often have stringent resource constraints, which are in conflict with the requirements of powerful machine learning models. Additionally, manually configuring sensing, tuning models and optimizing their embedded implementations is inefficient and labour intensive. To tackle those challenges, we propose and implement an automatic end-to-end framework *SMiLe* to co-design and optimize both sensing and machine learning. Our framework systematically examines trade-offs between sensing and model inference by navigating their joint design space and incorporating hardware-in-the-loop feedback, *i.e.*, latency and energy measurements, using a testbed. The benefits of our proposed framework are validated with a real-world industrial use case on motor health monitoring.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Data Analytics

General Terms

Design, Automation, Optimization

Keywords

Sensing, Machine Learning, AutoML, Design Automation

1 Introduction

The migration from cloud to edge analytics, where machine learning algorithms process the data locally on devices where it is generated, has experienced significantly increasing popularity [20]. Nowadays, there are already many important use cases in e.g. continuous environment/process monitoring, predictive maintenance and speech recognition [1], which perform all actions and computations locally. This often brings various benefits; *(i)* Cloud compute and data storage costs are reduced as processing is done on edge devices and data does not need to be constantly streamed to the cloud. *(ii)* For cases where real-time constraints are required, edge analytics is typically preferred as no communication latency to the cloud is incurred. *(iii)* If security is a critical concern as customers may not allow data to leave their premises, edge analytics becomes the only viable approach.

The benefits associated with edge analytics however also come with some major challenges. Edge computing platforms, in comparison to cloud computing infrastructures, are often severely limited in terms of computing and memory resources. While in the cloud one could easily deploy a big machine learning model to achieve good accuracy, this might simply not be feasible for edge devices as their resources are not sufficient to host large models. Therefore, ML models for edge devices must be accurate and tiny at the same time. By tiny, we mean their footprints in energy, latency and memory must be kept as small as possible. Due to this reason, this new paradigm of analytics is also referred to as *TinyML* [11].

Achieving high accuracy with a small system footprint is highly non-trivial. On the one hand, high accuracy of-

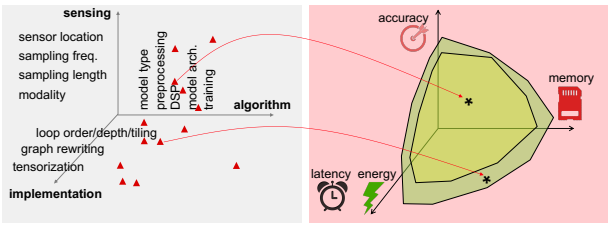


Figure 1: Challenges for edge analytics - *(i)* enormous design space covering sensing, machine learning and implementation, *(ii)* design constraints relating to latency, memory, energy consumption, and their complex trade-offs

ten implies complex ML models and heavy resource usage. Although small and accurate ML models could also exist for many practical problems, they are more difficult to identify [23]. On the other hand, the design space for edge analytics is complex and large. Prior works have mainly focused on optimizing machine learning regarding both models and their implementations, while neglecting the impact of sensing. In real-world deployments, sensing is an integral part of the processing pipeline; the amount of sensed data samples and the sensing frequency not only affects the energy consumption and latency of the sensing phase but also of the machine learning inference phase. For example, as we identified for an industrial bearing fault prediction use case, higher sensing frequency would actually help to reduce energy and latency required by machine learning models. To the best of our knowledge, the trade-offs between sensing and machine learning and their co-design have not yet been explored in the literature.

When designing efficient edge analytics solutions with the above outlined considerations, each of the three important steps (sensing, model architecture and implementation) requires specific decisions to be made. These decisions in return induce system level trade-offs of accuracy, memory, energy and latency, as illustrated in Figure 1. Navigating such a large design space manually while jointly optimizing multiple and often conflicting constraints (accuracy, memory, energy and latency) would not be feasible due to high development costs and difficulty to guarantee final solution quality.

Contributions. We propose and implement an end-to-end framework for edge analytics, which performs efficient design space exploration while generating edge analytics that jointly optimizes multiple system constraints. The contributions of this paper are as follows:

1. We design and implement an automatic firmware generation process based on Apache TVM [9] and Zephyr RTOS [2], which takes as input sensing and machine learning configurations and generates analytics executables in binary formats.
2. We introduce a hardware-in-the-loop (HIL) testbed for edge analytics to automatically benchmark energy and latency characteristics of both sensing and machine learning phases.
3. We propose an optimization engine based on AutoML with capabilities of parallel searching of edge analytics configurations and multi-objective optimization of accuracy, memory, latency and energy.
4. We integrate the above building blocks into a unified framework named *SMiLe* (automated end-to-end Sensing and Machine Learning co-design). The automatic end-to-end process optimizes the entire edge analytics processing pipeline covering both sensing and machine learning phases, and also explores the trade-offs and optimizes all important design considerations using live hardware-in-the-loop feedback.
5. We demonstrate with a real-world use case *SMiLe*'s capability in finding edge analytics with ∼5-10x better energy/latency characteristics while reducing search time by ∼90%. We further reveal with *SMiLe* interesting trade-offs between sensing and machine learning.

Outline. In Section 2, we discuss related work regarding AutoML, TinyML and some relevant use cases. We continue to motivate the work in this paper in Section 3 and present an overview of our solution in Section 4. Details about optimization for sensing and ML co-design are explained in Section 5. Furthermore, a detailed evaluation of our framework with a real-world use case is presented in Section 6. We conclude this paper in Section 7.

2 Related Work

Our proposed edge analytics framework takes inspiration from two major concepts, *i.e.*, AutoML and TinyML. By combining methods and technologies from both fields, we build an end-to-end sensing and machine learning co-design tool. Finally, we apply our approach to a real-world use case of classifying motor health conditions. We summarize the related work as follows.

AutoML. The rapid advancement of Deep Learning has led to numerous remarkable achievements in recent years. Various algorithms and models, particularly deep neural networks, have been successfully applied to solve complex problems in areas like natural language processing, computer vision and signal processing. Building these models however often not only requires extensive machine learning and subject-matter expertise but also resources for time intensive tasks like iterative fine-tuning and model optimization. The automated machine learning (AutoML) paradigm facilitates these requirements and efforts [14]. AutoML automates typical machine learning pipelines and its individual steps like data pre-processing, feature engineering, model architecture search, training and hyperparameter optimization. One particular area where AutoML has seen increasing research efforts is the automatic network architecture optimization to minimize memory requirements or inference latency. This can for instance be achieved by applying model compression [15]. Other approaches directly measure or approximate metrics like latency and integrate the measurements in the AutoML optimization, *e.g.*, by latency aware loss functions [7, 30], multi-objective optimization techniques [22, 3] or reinforcement learning [15, 28].

TinyML. The vision of Tiny Machine Learning (TinyML)

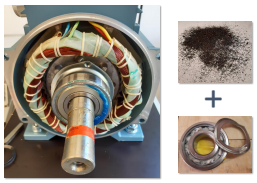
is to enable running machine learning models on embedded systems. These systems are typically limited in available resources like computing power, memory or energy [5] and, thus, pose additional challenges. TinyML therefore facilitates to move model inference from cloud to edge platforms and infrastructures like in typical IoT applications [10]. In order to achieve this computing shift, TinyML focuses on providing frameworks and methods to enable the implementation of machine learning models under various constraints of embedded systems. On one hand, specific interpreters and compilers for microcontrollers are required, like *Tensorflow Lite Micro* [11] or *TVM* [9]. On the other hand, neural networks need to be optimized in terms of their memory footprint, energy consumption and computational complexity to be fit for embedded systems. For instance, pruning [33] and quantization [13] are popular methods to simplify networks under different resource requirements.

AutoML + TinyML. Finally, due to the complex nature of optimizing machine learning models that can be run on embedded systems, there is strong research focus to combine concepts from both AutoML and TinyML [6]. Different works provide automated network architecture search methods, which not only optimize model accuracy but also different metrics imposed by the target hardware, such as, memory [12, 29, 19, 18], latency [4, 19, 18] and energy consumption [19] in a fully automated process.

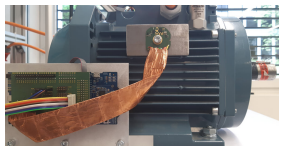
Our work proposes an end-to-end sensing and machine learning framework that uses an AutoML approach to implement and optimize models for embedded systems. In contrast to the existing work, our framework makes the following novel contributions.

1. *Sensing:* An optimal sensing set-up is an essential part of edge systems. Thus, our approach not only optimizes model parameters but also considers sensing, *i.e.*, the sampling frequency, as optimization target.
2. *Hardware-in-the-loop (HIL):* Different existing works incorporate hardware-dependent metrics like latency or energy consumption into the AutoML feedback loop. However, these metrics are often not directly measured on the target hardware but rather approximated, *e.g.*, by look-up tables for different model operations [4, 7, 18, 30] or hardware simulations [28]. Our approach directly measures all optimization objectives, *i.e.*, energy consumption, inference latency, memory requirements and model accuracy, using our testbed.

Motor Fault Prediction. Applying deep learning has become a popular approach to solve complex problems in the domain of smart manufacturing and especially machine health monitoring [27, 35]. Different works show how Convolutional Neural Networks (CNN) can be used on acceleration [34, 8] or current [16, 24] signals to accurately predict different faults of a motor. We use *SMiLe* to develop a model that predicts motor-bearing faults based on acceleration data. Using our framework we are able to not only optimize prediction accuracy like done in existing work but also target hardware dependent metrics.



(a) Controlled bearing damages by adding measured quantities of hardened metallic dust into the front motor bearing



(b) Accelerometer mounted on the side of the motor and interfaced to a Nordic nRF5340 development board

Figure 2: Motor bearing fault prediction setup

3 Motivation for *SMiLe*

In this section we demonstrate and motivate the need for automated sensing and machine learning co-design using a real-world use-case, *i.e.*, bearing fault analysis of motors. For this purpose, we built a motor testbed featuring three ABB M3AA asynchronous 3-phase motors. The bearings of the motors were damaged to different degrees by adding metallic dust into the bearing cases, *i.e.*, 0 mg for a healthy, 250 mg for a lightly damaged and 1000 mg for a heavily damaged bearing. Similar to related works [34, 8] we use a 3-axis accelerometer attached to the motor terminal block, see Figure 2. Our goal is to build a machine learning model using the acceleration data to predict the degree of damage in a motor by classifying it into 1 of the 3 metallic dust weight categories mentioned above. Furthermore, we do not only intend to optimize our model in terms of prediction accuracy but also in terms of energy consumption and inference latency when we implement it on an embedded system integrated into the motor. For this purpose, we ran the models on a Nordic nRF5340 SoC and measured latency and energy consumption using our testbed, see Sec. 4.4. For generating ML models to classify between bearings of different damage levels, we collected 40 minutes of data for each of the aforementioned bearing with our testbed, and trained classification models with PyTorch on a GPU server with GeForce RTX 2080 Ti. Note that for each specific sensing configuration, a new dataset is collected.

To highlight the impact of sensing and machine learning co-design and its automation, we compare three different settings. *(i) Baseline*: We manually setup and fix the sensing configuration, *i.e.*, sampling frequency and number of samples per window; a corresponding model is then manually constructed and trained on data collected with this particular sensing configuration. *(ii) Manual Sensing and ML Co-design*: We manually fine-tune the ML model at a fixed sensing frequency and a variable sensing window size to maximize accuracy and minimize number of model parameters. The resulting ML model is then run on the testbed using different sensing frequencies to minimize energy requirements and latency. *(iii)* SMiLe – *Automated Sensing and ML Co-Design*: Both sensing parameters and the ML design space and their trade-offs are automatically explored and the optimized edge analytics solution is generated with the use of our *SMiLe* framework.

In detail, for our baseline, similar to other studies [35, 24,

Table 1: Comparison between baseline (no optimization), manual sensing and ML co-design and *SMiLe* (fully automated approach) for an industrial bearing fault prediction use case

Optimizations	Search time (min)	Model	# Parameters	Sensing Frequency (Hz)	Sensing window	Accuracy	Sensing Energy (mJ)	Inference Energy (mJ)	Sensing Duration (ms)	Inference Duration (ms)
Baseline	2400	CNN1D	25601	416	500	1	11.787	11.996	1189	647
Manual sensing and ML co-design	5600	CNN1D	817	1660	150	0.998	1.41	0.165	80	9
SMiLe	480	CNN1D	65	1660	50	0.990	0.2978	0.0165	29.6	1.213

16], we use a CNN with 2 convolution layers with $(128, 64)$ neurons and a kernel size of 3, while the sensing frequency is pre-selected at 416 Hz. For manual sensing and ML co-design, we fine-tuned the baseline model to maximize the accuracy and minimize the number of parameters at a constant sensing frequency of 416 Hz and varying window size in $\{5, 10, 25, 50, 100\}$. We then exhaustively searched for sensing frequencies in $\{416, 833, 1660\}$ Hz to minimize the energy requirements and latency by using the same fine-tuned model. By performing manual sensing and ML co-design, we developed a final model with 2 convolution layers with $(15, 10)$ kernel sizes, and $(8, 8)$ number of channels. Lastly, for *SMiLe*, our automated sensing and ML co-design approach, we increased the search space in comparison to the manual approach (see Table 3) and our framework automatically navigates through this design space and found an optimized edge analytics pipeline model with 4 convolution layers with $(11, 7, 7, 7)$ kernel sizes and $(2, 2, 2, 2)$ number of channels.

We present the comparison of the above outlined approaches in Table 1. By comparing the baseline and the manual sensing and ML co-design, the benefits of co-designing both sensing and ML are shown: By increasing the sensing frequency to 1.66 kHz, the model size (number of parameters) is reduced by 96.8%, the window size (number of samples) required for each prediction is reduced by 70%, the sensing energy is reduced by 88% and the inference energy is reduced by 98.6%. This only incurs a drop of validation accuracy by 0.2%. By automating the co-design process with *SMiLe*, we can further improve the sensing and ML pipeline with a reduction of the model size by 92.04%, window size by 66.67%, sensing energy by 90% and inference energy by 63%. In comparison to the manual approach, *SMiLe* can find better solutions with an automated and intelligent search approach within a larger design space.

Table 1 also presents the development time required for different approaches. For manual exploration, we approximated the training and edge analytics generation time for each system configuration as 15 min based on trials for this use case; the final search time can be calculated as number of configurations tried multiplied by the time required for each configuration. For developing the baseline model, we explored 2 configurations of each model in $\{\text{LSTM}, \text{CNN1D}, \text{CNN2D}, \text{MLP}\}$, which are adopted by state-of-the-art solutions [35, 24, 16, 17, 32]; additionally, we choose a batch size from $\{256, 512, 1024\}$, epochs from $\{5, 10, 15\}$ and the learning rate from $\{0.01, 0.001\}$. Thus, the final search time is in total $\sim$ 2400 min ($2 \times 4 \times 3 \times 3 \times 2 \times 15$). Similarly, for manual sensing and ML co-design, we explored 25 configurations of the CNN1D model, with a window size in $\{5, 10, 25, 50, 100\}$ and the sensing frequency in $\{416, 833, 1660\}$ Hz, which resulted in a final search time of $\sim$ 5600min ($25 \times 5 \times 3 \times 15$). For *SMiLe*, we specify a total time budget of 480 min for automatic design space exploration. Finally, our framework is able to find solutions with better sensing and inference energy/latency while reducing the search time by 91.5%.

Our results above highlight that co-designing sensing and ML and automating this process with *SMiLe* enables to find optimized edge analytics solutions, which improve various constraints, e.g. energy, latency and model size. Furthermore, *SMiLe* automates the design process and delivers better results with significantly less development time. We believe the benefits by *SMiLe* are essential for deploying edge analytics on embedded systems, where both computing resources and development time are severely constrained.

4 Sensing and ML Co-Design with *SMiLe*

In this section we present an overview of our automatic end-to-end framework for edge analytics - *SMiLe* and explain its key features in detail.

4.1 Overview of *SMiLe*

A high-level overview of our framework is shown in Figure 3 and consists of three major building blocks: *(i)* a frontend (Sec. 4.2), *(ii)* a backend (Sec. 4.3) and *(iii)* a testbed (Sec. 4.4). Assuming a given edge analytics problem at hand, we first need to collect a dataset that is used as input to the frontend. Additionally, a configuration file specifies various system and experiment options, *e.g.*, regarding sensing, machine learning and the AutoML search routine. The frontend then constructs multiple machine learning pipelines based on the test configurations, trains the associated models and converts them to the ONNX format (Open Neural Network Exchange).

The models generated by the frontend are then transferred to the backend, which compiles them for a target system using Apache TVM [9]. The generated model code is then integrated with a corresponding sensing application, which is automatically generated with the help of the Zephyr kernel configuration system (Kconfig). Finally, the backend compiles multiple Zephyr applications for different sensing and machine learning configurations into corresponding executables.

The automatically generated edge analytics executables are pushed to our testbed via a REST API. During an exper-

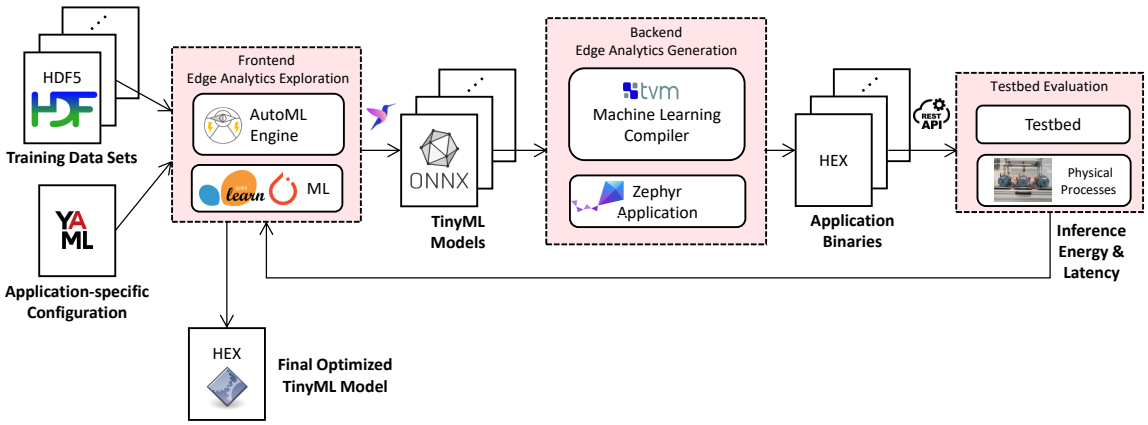


Figure 3: Overview of *SMiLe*: automatic end-to-end sensing and ML co-design

iment run, key characteristics like energy and latency during both sensing and inference phases are recorded and fed back to the frontend. Based on this feedback, the frontend continues to generate improved pipelines until either a given time budget is exhausted or an optimal solution is found.

In the following, we illustrate and present more details about the key building blocks for our framework *SMiLe*.

4.2 *SMiLe* Frontend

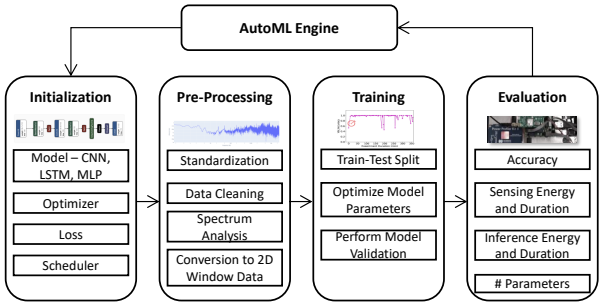


Figure 4: Overview of the *SMiLe* machine learning frontend

The main purpose of the frontend is to orchestrate the automated sensing and machine learning co-design, which is done by an extended AutoML engine based on *flaml* [26]. In the following section we describe the main components of the frontend as illustrated in Figure 4. We walk through how we utilize a training dataset and a configuration file to find an optimized sensing and machine learning pipeline.

4.2.1 *Initialization*

A *SMiLe* experiment can be set up by defining specific settings in the configuration file, according to which the sensing and ML pipeline is searched for an optimized solution. We divide the different settings into three categories based on their purposes. *(i) Sensing:* The most important sensing related settings are the choice of modalities, target sampling frequencies of the sensors if applicable and number of samples to collect for each prediction (ML inference). *(ii) Machine Learning:* These settings are a major part of the search space for the AutoML engine. In particular, we design pre-defined model templates allowing their own choices of parameters to tune, *e.g.*, neural network layer types, number of layers, channels and kernel sizes for convolutions, etc. We support both conventional models like decision trees and logistic regressions and neural networks like *e.g.*, multilayer perceptron (MLP), long short-term memory (LSTM) and convolutional neural networks (CNN). The framework is then able to not only select the best ML model type but also optimize the specific model architecture. Additionally, ML training configurations such as batch-size, learning rate and training epochs are supported by *SMiLe* as well. *(iii) AutoML:* Finally, the *SMiLe* frontend includes different AutoML search routine related settings, such as the optimization target, search methods and total time budget for searching.

4.2.2 *Pre-Processing*

Before training a model, the *SMiLe* frontend can selectively apply different pre-processing procedures (or none of them) to the input data. For example, the data can be scaled, *e.g.*, by z-score normalization to improve training convergence. Time-series related processing can be also applied with a sliding window. Conventional signal processing is also supported at this phase, *e.g.*, Fourier transformation or power spectral density analysis.

4.2.3 *Training*

During the training phase, the *SMiLe* frontend is constructing pipelines consisting of a sampling phase, a preprocessing phase and a machine learning model with settings sampled from the search space defined in the configuration file. To train the pipeline we first split the available data into training, validation and test datasets according to the defined split ratio in the configuration file. The core ML training process is carried out by Scikit-learn or PyTorch depending on the model type initialized.

4.2.4 *Evaluation*

In order to evaluate a trained model we collect four metrics: *(i)* The model **accuracy** is calculated using the validation dataset after training has been completed. *(ii)* The size

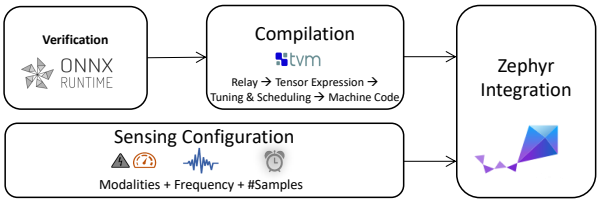


Figure 5: Overview of the *SMiLe* machine learning backend

of a model is characterized by the **number of model parameters** and is directly linked to the memory requirements on a target platform. *(iii)* Using our testbed we measure the **energy consumption** of a solution during its sensing and model inference phase. *(iv)* Similar to the energy consumption, the testbed is used to measure **latency** of sensing and model inference phases. These metrics are used by the AutoML routine to find an optimized pipeline iteratively.

4.2.5 *Frontend Output*

After the frontend has successfully found a pipeline it will generate *(i)* the trained model in ONNX and other formats (*e.g.*, PyTorch *.pth*) and *(ii)* the *.yaml* configuration file including specific settings that led to this solution. These outputs are passed to the *SMiLe* backend, which we will describe in the next section.

4.3 *SMiLe* Backend

We continue to present the *SMiLe* backend, which takes the high level model description from our frontend, compiles it into C code and integrates it with a sensing application to form a complete edge analytics pipeline, as shown in Figure 5.

4.3.1 *Model Verification*

Before compiling ML models to edge analytics binaries, *SMiLe* backend allows to quickly visualize and validate models on an easy-to-deploy platform like a CPU or GPU. To this end, we make use of the ONNX Runtime; in short, it parses model information and data, builds an in-memory computation graph and then maps this graph to the underlying hardware (CPU or GPU) to perform model inference. To verify the correctness of an exported ONNX model, our backend takes both ML models (PyTorch and ONNX) and their corresponding test data; the computed outputs on the test data by the ONNX runtime are checked against ground truths to validate models before we convert them to embedded implementations. Note during the automatic design space exploration of *SMiLe*, model verification is optional and we manually validate the best found models.

4.3.2 *Model Compilation and Executable Generation*

Machine learning compilers are emerging as a standard software infrastructure to convert high level machine learning models into low level implementations for different processor architectures, *e.g.*, X86, CUDA, RISC-V and ARM. We chose Apache TVM as our ML compiler as it has dedicated support for microcontrollers in comparison to *e.g.*, Intel PlainML, Google XLA and Facebook Glow; such support is critical for many edge computing platforms.

With the help of TVM, our backend first builds up a graph representation of the ML model and then performs graph level optimization like common sub-expression elimination, operator fusion, dead code elimination, etc. Next, operator level optimization and code generation is performed. This is done by lowering the tensor expressions of individual operators by taking into account the hardware characteristics to form a feasible schedule. The outputs are low level tensor intermediate representations (TIR), which then are lowered to standard compiler immediate representations (IRs, *e.g.*, LLVM IR); finally by connecting with an existing compiler like LLVM (for lowered LLVM IR) final code generation is performed.

To target microcontrollers, we adopt specifically MicroTVM and its ahead-of-time (AoT) compilation mode to generate a minimal machine learning runtime for a given machine learning model. The generated embedded runtime contains all necessary C code needed to perform inference, *e.g.*, parameters, model operators, graph level computation, and memory management, etc.

4.3.3 *Sensing Integration*

The final executable also contains the sensing task/phase, which is automatically modified by the *SMiLe* backend based on system configurations (*e.g.*, frequency and number of sensed samples) and merged with the machine learning runtime generated by TVM as well as Zephyr RTOS to form the final firmware for the edge analytics pipeline. We leverage kernel configuration as provided by Zephyr to automate sensing configuration and its integration with ML. While we use Zephyr for our proof-of-concept, our approach generates plain C code and is therefore agnostic of the specific operating system used on the embedded device.

4.4 Testbed Infrastructure

We employ an automated testbed infrastructure to obtain measurements of energy and latency when executing the resulting sensing and ML pipeline on the embedded device. The architecture of our testbed is depicted in Figure 6 and consists of three parts: (1) the infrastructure part consists of a software-based testbed controller and a hardware-based testbed observer, (2) the embedded device, and (3) the application-specific instrumentation and control (*e.g.*, the control of an AC asynchronous motor using a variable speed drive).

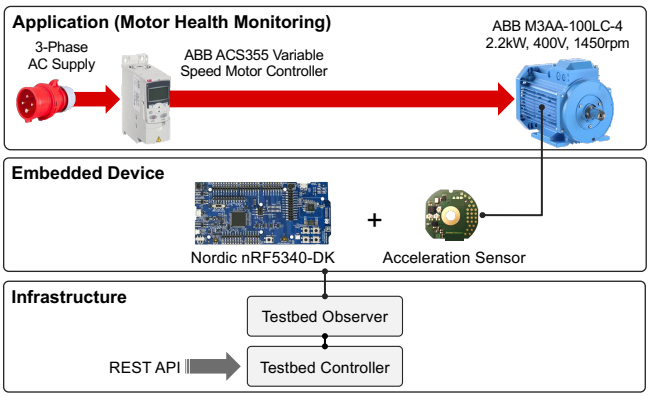


Figure 6: Architecture of a motor testbed used to evaluate *SMiLe* for the purpose of motor bearing health analysis

4.4.1 Testbed Observer and Controller

The *testbed observer* controls the execution of a test on the embedded device. It is a custom hardware and software platform based on the Raspberry Pi4 single-board computer running Linux, as shown in Figure 7. The embedded device under test is supplied with power through the Nordic PPK2 power profiler, which is at the same time also measuring the current drain of the embedded device at a sampling rate of 10 kSamples/sec. Furthermore, the PPK2 also monitors several GPIO signals of the embedded device, which indicate whether the embedded device is in idle, sensing or inference state. The serial output of the embedded device under test is also logged to record the results of the inference phase. At the beginning of each test, the firmware image is written to the flash memory of the nRF5340 microcontroller using the NXP MCU-Link Pro debug probe.

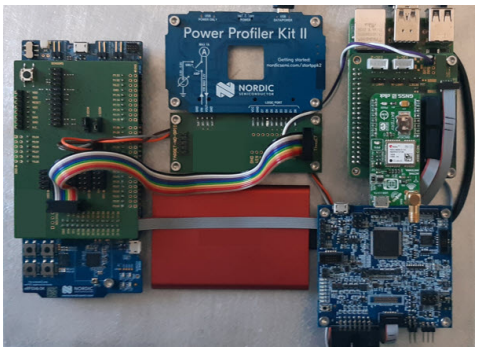


Figure 7: Testbed observer based on a Raspberry Pi4 interfaced to commercial power profiler, JTAG programmer and a Nordic nRF5340 development kit

The *testbed controller* is a collection of software services running on a server machine connected to the same IP network as the observers. The controller is responsible to schedule the execution of a test on the observers. Furthermore, measurement data collected by the observers are stored in a time-series database at the controller. A REST-based API is used to submit jobs to the testbed controller as well as for fetching of measurement results (latency, engery consumption).

4.4.2 Embedded Device: Nordic nRF5340-DK

We employ the Nordic nRF5340 development kit (nRF5340-DK) as our target platform for sensing and inference. It is based on the nRF5340 dual-core microcontroller, which combines an application and network processor into a single package. For our use case, we make use of the application core only, while the network core remains disabled during our tests. We have built a custom sensor board, which can be attached to the outside frame of the motor to measure vibration. The sensor board features the ST ISM330DLC 3-axis digital accelerometer and gyroscope, which is connected using the serial peripheral interface (SPI) bus to the microcontroller. Note that the nRF5340-DK is operated in the *nRF only* mode, which keeps the nRF5340 SoC disconnected from the on-board debugger and interface circuitry, in order to provide accurate current measurements of the SoC only. We use three dedicated GPIO output pins to indicate the current operating mode (sensing, inference or idle).

4.4.3 Application (Motor Health Monitoring)

The third component of our testbed architecture is specific to the application scenario. In our case, we use a three-phase asynchronous industrial motor (ABB M3AA-100LC-4) controlled by a variable speed drive (ABB ACS355). The controller of the drive is interfaced by Modbus/TCP to the testbed observer, which allows us to start and stop the motor, as well as control its direction and speed from the testbed observer during a test.

5 Edge Analytics Optimization with *SMiLe*

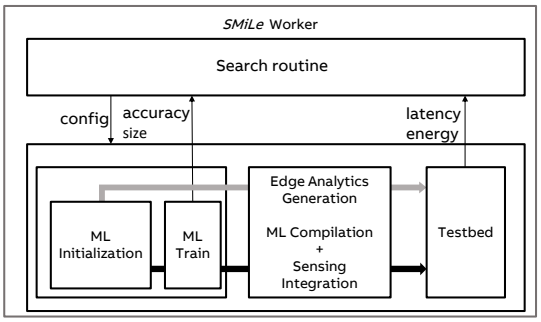
We dive deeper in this section into *SMiLe* to show how optimization of edge analytics is performed. To this end, we first give a brief overview of *SMiLe* and then present details of the search routine used in *SMiLe*.

5.1 Optimization Overview

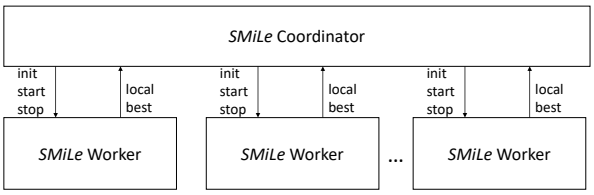
With the integration of *SMiLe* frontend, backend and our testbed, we can generate for each input dataset and system configuration a corresponding sensing and ML pipeline. To search for optimized edge analytics co-optimizing accuracy, energy, latency and memory, we use a search routine (AutoML engine in our case) to drive the entire end-to-end process, see Figure 8a. This works as follows:

1. The system initiates and configures the search space from YAML specifications (and dedicated ML templates if specified).
2. We sample one system configuration and try the configuration through our entire frontend and backend. Sensing configuration includes modalities, number of samples and frequency while ML configuration includes preprocessing options, feature selection, signal processing, model type and corresponding hyperparameters.
3. We initialize a machine learning model from the sample space. After this the untrained model is directly fed into our backend and testbed to get measurements like latency and energy. This is because such metrics are irrelevant to training, which only affects the accuracy of the model.
4. While our backend and testbed are experimenting with the sampled model, our frontend trains the model in parallel to find suitable model parameters. The final model accuracy and parameters are recorded.
5. Our search routine combines results from both frontend and backend (e.g. accuracy, number of model parameters, latency and energy) into a global metric to jointly consider all those concerns. After this, if desired solution quality is achieved, the process is terminated; otherwise, we select the next system configuration which would improve the overall solution and go to step 3.

Note that the workflow we presented above is for *SMiLe*'s design space exploration phase, during which ML training and edge analytics generation (followed by testbed benchmarking) run in parallel to improve efficiency. For final deployment, edge analytics will be generated for trained and



(a) End-to-end edge analytics optimization in *SMiLe*



(b) Parallel *SMiLe* workers and global coordinator

Figure 8: Optimization of *SMiLe*

optimized ML pipelines, as indicated by the bottom execution path in Figure 8a.

To further speedup the end-to-end process, we launch multiple parallel learners (Figure 8b), each of which is performing operations as shown in Figure 8a. Those learners are independent in searching for good system configurations (sensing and ML) and contribute together to find good quality solutions. To coordinate learning between different learners, we use a global coordinator, which is responsible for monitoring the learning performance of individual learners. It will prioritize better performing learners to be run on the limited hardware resources and stop learners with poor performance in order to launch new ones.

5.2 Search Routines in *SMiLe*

We proceed to present more details about search routines used in *SMiLe* and how we perform multi-objective optimization balancing and co-optimizing different design goals, *e.g.*, accuracy, model size, energy and latency.

5.2.1 AutoML

We adopt an AutoML engine to automatically search and optimize the entire design process covering both sensing and machine learning. Conventionally, AutoML has been adopted to create machine learning pipelines alone, e.g. preprocessing, feature engineering, model selection and hyperparameter optimization. In this paper, we extend AutoML in two different manners. Firstly, we include searching of sensing configurations into the AutoML engine, such that sensing is optimized jointly with machine learning. Secondly, we incorporate feedback from our IoT testbed, *i.e.*, latency and energy for both sensing and ML, into the AutoML process. This allows the AutoML engine to iteratively find improved edge analytics configurations.

Optimization in *SMiLe* is built based on *flaml* - a library for automated machine learning & tuning developed by Microsoft [25]. *SMiLe* requires an optimization target while searching for the best configurations. The optimization target is to maximize accuracy in the conventional sense; we extend it to also include considerations of model size, duration/energy for both inference and sensing phases, etc. It then searches across the entire design space (all possible system configurations) to automatically optimize the specified target. During this process, *SMiLe* is able to leverage the advantages of both local search routines like CFO [31] (*SMiLe* workers) to fine tune a proposed system configuration and a global search routine with Bayesian optimization (*SMiLe* coordinator) to orchestrate local search routines (e.g. starting configurations, prioritize/stop local search routines). To speedup searching, *SMiLe* adopts a distributed execution framework Ray [21] and parallelizes global and local search routines. In detail, optimization in *SMiLe* works as follows:

1. At the start, *SMiLe* initializes one global search routine and the first local *SMiLe* worker with configurations based on suggested low cost ones by the user.
2. *SMiLe* then uses a selector to find the best search routine to run among available ones based on a priority metric. This metric measures the projected performance (improvement on multi-objective loss) of a search routine in a fixed amount of time based on its past trajectory.
3. *SMiLe* continues to execute the selected search routine, which will propose a new system configuration, train the edge analytics and evaluate it on our IoT testbed. Time required and performance achieved by this search routine are recorded. If the system configuration is proposed by the global search routine, then *SMiLe* checks if the performance is better than that of at least half of the existing ones; in case this is true and there are hardware resources available, *SMiLe* initializes a new *SMiLe* worker with the latest configuration proposed by global search routine.
4. Subsequently, *SMiLe* does bookkeeping and removes converged routines and those with similar performance and configurations to existing ones.
5. *SMiLe* updates the global search routine's model based on the configurations tried and performance achieved.
6. Steps 2-5 are repeated until a time budget is exhausted.

5.2.2 Multi-Objective

SMiLe requires a multi-objective target for optimization in order to jointly optimize accuracy, energy, latency and size of the generated edge analytics. This target is also used for prioritizing different search routines as mentioned in Sec. 5.2.1. To this end, we extended the AutoML engine used in *SMiLe* to perform multi-objective optimization by using a linear combination of different design considerations, each of which is multiplied with its specific weight. In this setting, one metric with larger value after scaling/multiplication has more impact on the outcome of the linear function; as a result, such a metric will be "favored" more during optimization.

In detail, assume that we have a list of metrics M to optimize. To derive the weight w_m of one metric $m \in M$, we

Table 2: Metric weights for bearing fault prediction

Metric	Range	Normalizing Factor	Priority Factor	Weight
Validation Accuracy	[0, 1]	1	10^6	10^6
#Parameters	[30, 24109]	10^{-5}	10^5	1
Sensing Energy	[1401, 470994]	10^{-6}	10^5	10^{-1}
Sensing Latency	[0.028294, 11]	10^{-2}	10^5	10^3
Inference Energy	[43, 9813]	10^{-4}	10^4	1
Inference Latency	[0.000742, 0.100]	10^1	10^4	10^5

first normalize the values of each metric by conducting a test run of *SMiLe*. Scales of considered metrics like accuracy, latency and energy are returned by *SMiLe* and we subsequently normalize them to the same scale by multiplying each metric value m with a corresponding normalizing factor n_m. We then decide the relative importance of each metric based on trade-offs we want to achieve for the specific use case; to reflect this in the final optimization, we assign a priority factor p_m to each metric m. The final weight of each metric m, w_m, is then calculated as the product of the corresponding normalizing factor and priority factor: $w_m = n_m \times p_m$. Consequently, the optimization target (global metric) used by *SMiLe* is $\sum_{m \in M} w_m \times m$.

For the real-world use case explained in Sec. 3, we summarize both normalization and priority factors used for 6 considered metrics in Table 2. To first derive normalizing factors, we scale the metrics such that their maximums are mapped to the range $[0,1]$. We then assign a priority factor to each group of metrics with priority factors reduced by 10 for each subsequent metric group. Note that one can choose different priority factors based on the concrete use case.

6 Experimental Evaluation

In this section we analyze and evaluate various aspects of *SMiLe* using the previously described bearing fault analysis as our real-world test case. In Sec. 6.1 we present how *SMiLe* searches for an optimal edge analytics pipeline among a large design space. Finally, in Sec. 6.2 we discuss different system level related trade-offs, e.g. energy consumption and latency.

6.1 Design Space Exploration with *SMiLe*

We start with describing our experimental setup in Sec. 6.1.1 and proceed with showing how *SMiLe* explores the search space during an experiment run in Sec. 6.1.2.

6.1.1 Setup

Data. To evaluate *SMiLe* we use the bearing fault analysis problem as explained in Sec. 3 and Sec. 4.4. We first collected a dataset using our motor testbed featuring three motors with differently damaged bearings. These motors were run at different RPM's and directions. Thus, we collected accelerometer data for each of the 3 motors running at $\{4, 19, 37, 55, 94, 148, 250, 535, 994, 1500\}$ RPM and in $\{\text{Forward}, \text{Reverse}\}$ direction. For each motor, speed and direction combination we collected 2 minutes of data sampled at 1660 Hz. Thus, the final dataset consists of $3 \times 10 \times 2 \times 120 \times 1660 \approx 12 \cdot 10^9$ samples. We solely used samples from the y-axis of the accelerometer, which captures the highest variance in movement when the motor is running. The sensor data streams were then split into individual windows using a sliding window approach, where the size of the window is optimized. We then used 80% of the data for training, 10% for validation and 10% for testing the final models.

Experiment. We analyze two different bearing fault analysis problems, *i.e.*, *(i)* 3-class classification to distinguish between a healthy (0 mg dust), lightly (250 mg) and heavily (1000 mg) damaged motor and *(ii)* binary classification to distinguish between a healthy and a heavily damaged motor. For each of the two problems we ran a *SMiLe* experiment for 12 hours. In the remainder of this section we summarize the results of these experiments for both problems.

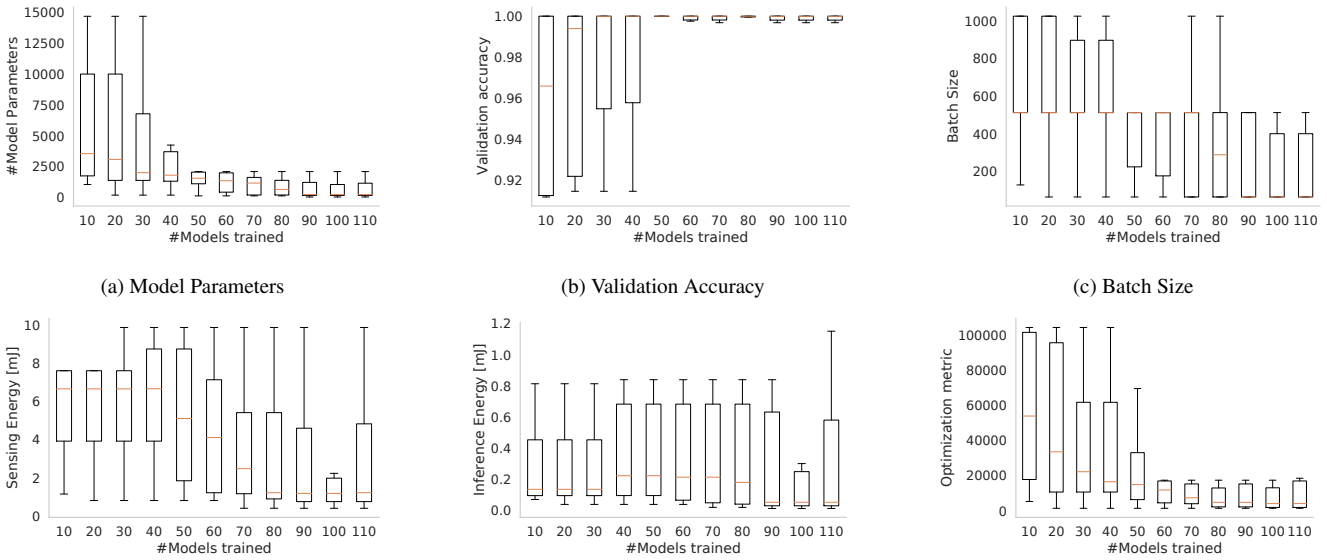
Table 3: Design space for bearing fault prediction

Hyperparameters	Range	Category
Window size	[25, 700]	Sensing
Sampling Frequency	[12.5, 26, 52, 104, 208, 416, 833, 1660]	Sensing
Batch size	[2, 4096]	ML
Epochs	[1, 20]	ML
Learning Rate	[0.0001, 0.1]	ML
Model Architecture	MLP, CNN1D, LSTM, CNN2D	ML
# of Layers	[1, 10]	ML
Hidden Size	[2, 512]	ML
# of Channels	[1, 20]	ML
Kernel Size	[1, 50]	ML

Design Space Configuration. To find the best model configuration, *SMiLe* needs to search a large design space by optimizing various parameters. Table 3 lists all parameters, their ranges as well as categories, *i.e.*, sensing or machine learning (ML). Machine learning parameters include training-related hyper-parameters and model architecture configurations. These have a direct impact on model accuracy and model size and, thus, also on inference latency and energy consumption, which we show in Sec. 6.2. Sensing parameters include configurations that affect the data collection process. Their optimization naturally impacts the sensing latency and energy consumption as highlighted in Sec. 6.2. Note that we downsample the data by skipping samples if a sensing frequency below 1660 Hz is utilized.

6.1.2 Exploration

In order to understand how *SMiLe* operates and searches the large design space, we examine the evolution of different metrics over time during an experiment. Figure 9 shows the number of model parameters, validation accuracy, batch size, sensing energy, inference energy and the global metric at different times during an experiment run. Each boxplot summarizes the statistics of the metric over the 10 latest models after a certain amount of models have been trained overall. Within 12 hours *SMiLe* trained and benchmarked



(a) Model Parameters (b) Validation Accuracy (c) Batch Size

(d) Sensing Energy (e) Inference Energy (f) Global Metric

Figure 9: Evolution of various metrics during *SMiLe* design space exploration (indicated by the number of fully trained models)

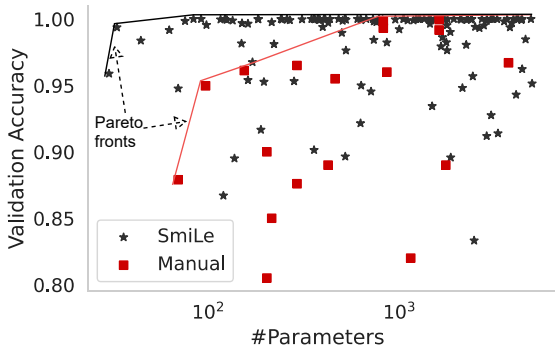


Figure 10: Validation accuracy vs. number of model parameters with Pareto front in the top-left corner

110 different pipelines. Figure 9a shows how in the early stages of the experiment *SMiLe* evaluates large ML models and gradually reduces the number of parameters in order to end-up with notably smaller models. Additionally Figure 9b and Figure 9c highlight how the accuracy increases and the batch-size decreases over the course of an experiment. These observations show how *SMiLe* initially explores the design space in a broad manner, then continues to fine-tune the models with just sufficient amount of data. These results are also reflected in Figure 10, which shows validations accuracy versus number of model parameters for various trained models. Compared to manually fine-tuning the models, which we describe in Sec. 3, *SMiLe* is able to improve the Pareto front and explore more accurate and smaller models. Additionally, these results were achieved with an experiment in 12 hours, which is considerably faster than manually fine-tuning. Finally, both the sensing (Figure 9d) and inference energy consumption (Figure 9e) as well as the final global metric (Figure 9f) are optimized, *i.e.*, decreased, over time.

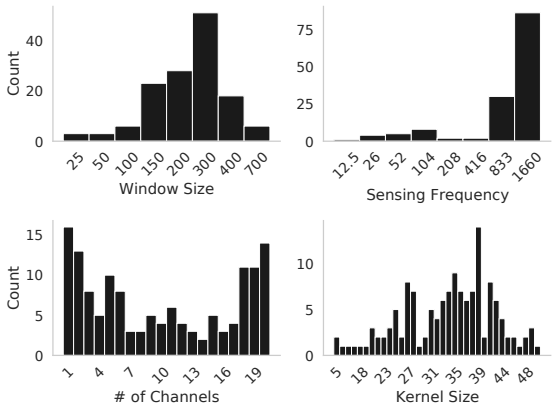


Figure 11: Counts of how many times specific settings are explored by *SMiLe* during an experiment

Figure 11 shows how *SMiLe* explored different design space configurations over the course of the experiment. We present here several representative design parameters from Table 3, and observe that *SMiLe* will quickly disregard settings, which will not lead to an optimized solution, *e.g.*, lower sensing frequencies. Settings that finally lead to the best performing solution are however more frequently tested and explored.

At the end of the experiment, *SMiLe* provides us with the configurations and results of the optimized edge analytics pipeline. For both the binary and 3-class classification prob-

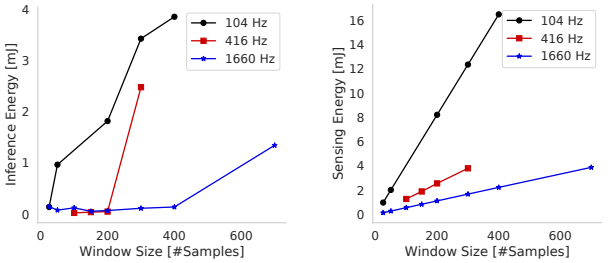
lems, the test accuracy was around 99.8% for a CNN model with a size of 256 and 627 parameters respectively. The optimal sensing frequency is 1660 Hz and the window size 50 samples for the binary classification and 300 samples for the 3-class classification.

6.2 System Level Trade-Offs

We continue to present system level trade-offs between sensing and machine learning as identified by *SMiLe*.

6.2.1 *Energy*

The energy consumption of an edge analytics pipeline is greatly influenced by various design choices. In the following we analyze how particularly the window size and sensing frequency affect the overall energy consumption.

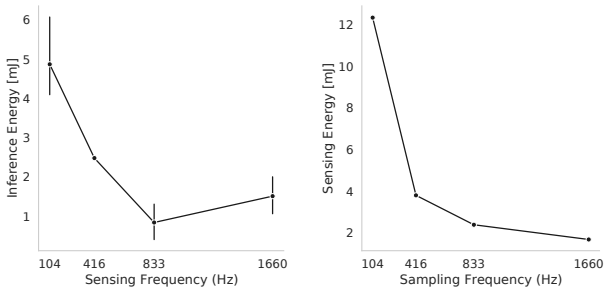


(a) Inference energy (b) Sensing energy

Figure 12: Impact of window size on inference and sensing energy consumption at different sensing frequencies

Window Size. Figure 12a and Figure 12b show the impact of the window size on the inference and sensing energy consumption resp at difference sampling frequency. As expected, the required energy for sensing increases with the size of the window at a fixed sampling frequency. The inference energy however typically depends on the model size. We observe that the inference energy is minimized at small window sizes and high sampling frequencies. Thus, *SMiLe* was able to find well performing models that solely require samples in short time windows, that contain sufficient and relevant information, to determine the health of a motor while minimizing the overall energy consumption. This shows that it is of high importance to search for the appropriate window size during optimization.

Sensing frequency. Similar results can be observed when inspecting the impact of the sampling frequency on the energy consumption. Figure 13 shows this relationship for both the inference and sensing phases. As expected and also described before, the sensing energy consumption purely depends on the window size and sampling frequency and, thus, decreases with an increasing sampling frequency at a fixed window size as shown in Figure 13b. An interesting observation is shown in Figure 13a, where the inference energy is the smallest at a sampling frequency of 833 Hz. We hypothesis this frequency allows to capture most relevant information for bearing fault prediction; consequently, it will also be easier to find smaller ML models to classify correct bearing fault classes. Note that although not shown, the same behaviour also appears at different window sizes. This again



(a) Sensing frequency vs. inference energy (b) Sensing frequency vs. sensing energy

Figure 13: Impact of sensing frequency on inference and sensing energy at a fixed window size of 300 samples

shows that sensing related parameters have a clear impact on the overall energy consumption and, therefore, it is important to also include them in the search space to find right trade-offs between sensing and machine learning.

6.2.2 *Latency*

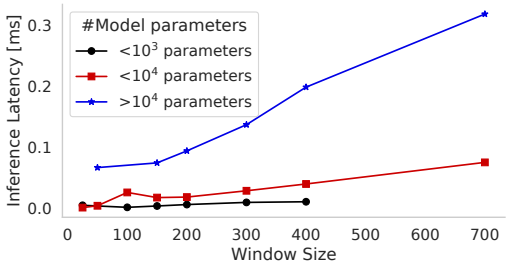


Figure 14: Window size vs. inference latency and number of model parameters

Another important aspect of edge analytics is latency, *i.e.*, the sensing latency on one hand, which is the time it takes to collect a window of samples, and the inference latency on the other hand, which is the time it takes for a model to produce a prediction after we provide input samples. The sensing latency is directly affected by the window size and the sampling frequency, which can be calculated as $\frac{window\ size}{sampling\ frequency}$. The inference latency however is greatly impacted by two factors: *(i)* The size of the machine learning model, meaning the larger a model the more time it requires to calculate a prediction and *(ii)* the window size, *i.e.*, the input to the model. This relationship is also shown in Figure 14. Especially for complex problems that require both a large model for accurate predictions and short inference latencies due to time critical decisions it is essential to find an optimized pipeline, which can be achieved by our *SMiLe* framework. Note that naturally the inference latency also correlates to the inference energy consumption, hence, the model size also impacts the inference energy.

7 Conclusion

We present in this paper *SMiLe* - an automatic framework for edge analytics which optimizes and co-designs both sensing and machine learning. The proposed framework consists of three critical components. First, our frontend covers a conventional ML pipeline from the data science point of view, which takes a dataset as input and generates trained ML models. Second, our backend takes generated analytics from the frontend, automatically converts them into embedded implementations together with desired sensing configurations. Finally, the entire frontend and backend are integrated together and we adopt an effective search routine to automatically find efficient embedded analytics, which are balancing accuracy, latency, energy and memory requirements. This multi-objective optimization is made feasible due to integration of our framework with our proposed testbed, which automatically benchmarks embedded applications in terms of their latency and energy characteristics with hardware-in-the-loop. *SMiLe* is demonstrated with a real-world use case, while showing the capability to guarantee great accuracy and to reduce system footprints significantly in terms of energy, latency and memory.

8 References

[1] Leveraging Industrial IoT and advanced technologies for digital transformation. https://tinyurl.com/yc398xfe. Accessed: 2022-04-29.

[2] Zephyr project. https://zephyrproject.org/. Accessed: 2022-04-29.

[3] D. Aquino-Brítez, A. Ortiz, J. Ortega, J. León, M. Formoso, J. Q. Gan, and J. J. Escobar. Optimization of deep architectures for eeg signal classification: An automl approach using evolutionary algorithms. *Sensors*, 21(6):2096, 2021.

[4] C. Banbury, C. Zhou, I. Fedorov, R. Matas, U. Thakker, D. Gope, V. Janapa Reddi, M. Mattina, and P. Whatmough. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of Machine Learning and Systems (MLSys)*, 3, 2021.

[5] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, et al. Benchmarking tinyml systems: Challenges and direction. *arXiv preprint arXiv:2003.04821*, 2020.

[6] H. Cai, J. Lin, Y. Lin, Z. Liu, H. Tang, H. Wang, L. Zhu, and S. Han. Enable deep learning on mobile devices: Methods, systems, and applications. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 27(3), 2022.

[7] H. Cai, L. Zhu, and S. Han. Proxylessnas: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations (ICLR)*, 2018.

[8] C.-C. Chen, Z. Liu, G. Yang, C.-C. Wu, and Q. Ye. An improved fault diagnosis using 1d-convolutional neural network model. *Electronics*, 10(1), 2020.

[9] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[10] L. Cui, S. Yang, F. Chen, Z. Ming, N. Lu, and J. Qin. A survey on application of machine learning for internet of things. *International Journal of Machine Learning and Cybernetics*, 9(8), 2018.

[11] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang, et al. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems (MLSys)*, 3:800–811, 2021.

[12] I. Fedorov, R. P. Adams, M. Mattina, and P. Whatmough. Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers. *Advances in Neural Information Processing Systems (NeurIPS)*, 32, 2019.

[13] S. Ghamari, K. Ozcan, T. Dinh, A. Melnikov, J. Carvajal, J. Ernst, and S. Chai. Quantization-guided training for compact tinyml models. In *Research Symposium on Tiny Machine Learning*, 2020.

[14] X. He, K. Zhao, and X. Chu. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2021.

[15] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European conference on computer vision (ECCV)*, 2018.

[16] T. Ince, S. Kiranyaz, L. Eren, M. Askar, and M. Gabbouj. Real-time motor fault detection by 1-d convolutional neural networks. *IEEE Transactions on Industrial Electronics (TIE)*, 63(11), 2016.

[17] B. Li, M.-Y. Chow, Y. Tipsuwan, and J. Hung. Neural-network-based motor rolling bearing fault diagnosis. *IEEE Transactions on Industrial Electronics*, 47(5), 2000.

[18] E. Liberis, Ł. Dudziak, and N. D. Lane. μnas: Constrained neural architecture search for microcontrollers. In *Proceedings of the 1st Workshop on Machine Learning and Systems (EuroMLSys)*, 2021.

[19] J. Lin, W.-M. Chen, Y. Lin, C. Gan, S. Han, et al. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems (NeurIPS*, 33:11711–11722, 2020.

[20] F. Liu, G. Tang, Y. Li, Z. Cai, X. Zhang, and T. Zhou. A survey on edge computing systems and tools. *Proceedings of the IEEE*, 107(8):1537–1562, 2019.

[21] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging ai applications, 2017.

[22] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2820–2828, 2019.

[23] M. Tan and Q. V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. 2019.

[24] R. N. Toma, F. Piltan, and J.-M. Kim. A deep autoencoder-based convolution neural network framework for bearing fault classification in induction motors. *Sensors*, 21(24), 2021.

[25] C. Wang and Q. Wu. FLO: fast and lightweight hyperparameter optimization for automl. *CoRR*, abs/1911.04706, 2019.

[26] C. Wang, Q. Wu, M. Weimer, and E. Zhu. Flaml: a fast and lightweight automl library. *Proceedings of Machine Learning and Systems*, 3:434–447, 2021.

[27] J. Wang, Y. Ma, L. Zhang, R. X. Gao, and D. Wu. Deep learning for smart manufacturing: Methods and applications. *Journal of Manufacturing Systems*, 48, 2018.

[28] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han. Hardware-centric automl for mixed-precision quantization. *International Journal of Computer Vision (IJCV)*, 128(8):2035–2048, 2020.

[29] T. Wang, K. Wang, H. Cai, J. Lin, Z. Liu, H. Wang, Y. Lin, and S. Han. Apq: Joint search for network architecture, pruning and quantization policy. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[30] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10734–10742, 2019.

[31] Q. Wu, C. Wang, and S. Huang. Frugal optimization for cost-related hyperparameters. In *AAAI'21*, 2021.

[32] R. Yang, M. Huang, Q. Lu, and M. Zhong. Rotating machinery fault diagnosis using long-short-term memory recurrent neural network. *IFAC-PapersOnLine*, 51(24):228–232, 2018. 10th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes SAFEPROCESS 2018.

[33] T.-J. Yang, Y.-H. Chen, and V. Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[34] H. Zhang, B. Ge, and B. Han. Real-time motor fault diagnosis based on tcn and attention. *Machines*, 10(4), 2022.

[35] R. Zhao, R. Yan, Z. Chen, K. Mao, P. Wang, and R. X. Gao. Deep learning and its applications to machine health monitoring. *Mechanical Systems and Signal Processing*, 115, 2019.