Exploring Cloud Assisted Tiny Machine Learning Application Patterns for PHM Scenarios

Xingyu Zhou[1], Zhuangwei Kang[2], Robert Canady[3], Shunxing Bao[4], Daniel Balasubramanian[5], and Aniruddha Gokhale[6]

[1,2,3,4,5,6] *Dept of CS, Vanderbilt University, Nashville, TN 37235, US*
{*xingyu.zhou, zhuangwei.kang, robert.canady, shunxing.bao, daniel.a.balasubramanian, a.gokhale*}@vanderbilt.edu

ABSTRACT

Given the diverse deployments of sensor nodes in prognostics and health management (PHM) applications, the use of small form-factor, low-cost and power-efficient microcontrollers (MCUs) has become a practical option for long-term monitoring and front-end data-processing. Hardware advances have enabled small MCU devices to run light-weight machine learning (especially deep neural networks) thereby enabling inference tasks using tiny machine learning (Tiny ML) models executing closer to the data source sensors. Although TinyML like approaches have previously been proposed for some cases in PHM, existing approaches have mainly targeted PHM applications that use single data sources and case-specific models as opposed utilizing prediction models trained from general machine learning frameworks and requiring fusion of multiple distributed data sources. Unfortunately, pure MCUs lack the capacity to conduct such analytics. This work aims to address these limitations by using TinyML deployed at the edge in cooperation with system-level machine learning executing in the cloud. Specifically, we study applications in which sensor data is collected and used to predict system health status and perform remaining useful life regression. We also show how edge MCU devices and cloud computing can be combined and adapted to satisfy diverse requirements, such as latency, power and communication. We also describe the limitations of the current MCU-based deep learning in data-driven prognostics. To the best of our knowledge, this is the first work to systematically investigate the TinyML-Cloud cooperation for data-driven prognostics. We target this as a vision paper and aim to provide a high-level guideline for future PHM application designs involving smart MCU-based decision making.

1. INTRODUCTION

With the current development of machine learning, especially deep learning, statistical prediction models like neural networks have achieved state-of-the-art performance for many tasks pertaining to cyber-physical systems including prognostics and health management (PHM). Recent hardware advances have made it possible to run small but optimized machine learning models on low-power microcontrollers (MCUs), such as Arduino or STM32, at the edge. By being closer to the source of the data, PHM applications can be more efficient. This concept is called embedded machine learning, or TinyML (Warden & Situnayake, 2019). By providing on-device sensor data monitoring and processing, TinyML provides promising opportunities for PHM applications in early warning generation and smarter decision making at low-power as well as low cost with a low-latency guarantee.

Executing edge-based machine learning models for PHM applications has been studied in the literature (Grosvenor & Prickett, 2011). Past works usually use MCUs for condition monitoring and set relative simple threshold strategies for early warnings (Song, Luan, Shi, Li, & Wang, 2020). These past works show the potential of MCU-based smart decision making for PHM applications (Farinholt et al., 2018). At the same time, they also show one general difficulty of using MCUs for data-driven PHM because the prediction models are usually case-specific and lack general applicability. Moreover, the limited hardware capacity of the MCUs and high data volumes of PHM applicatons makes it hard to rely on MCUs alone for complete PHM capabilities.

Furthermore, efficiently assigning prediction or inference tasks to TinyML devices for PHM applications is difficult due to the following challenges:

- PHM application data sources are often from different sensors. It is difficult to fuse sensors when we use them as data inputs for TinyML models.
- PHM applications often involve systems with manual configurations. We need to consider how these configurations would be used as input features of TinyML models in a proper way.
- PHM applications often make use of prediction models with spatio-temporal considerations. Time synchronization and sensor data transfer would become a system-level

challenge for sensor networks with limited hardware resources at the sensor end.

- Recent PHM applications involve higher data volumes at the sensor end e.g., live video or audio data. Efficient model training and optimization for potential TinyML deployments has become a challenge.

To address these challenges, we propose the use of cloud computing as a high-level support system that can work in conjunction with TinyML deployments. In this paper, we explore potential application patterns where MCUs and cloud computing can be adapted to different kinds of machine learning models and combined in flexible ways thereby catering to diverse PHM application requirements. Overall our work emphasizes the following critical aspects of TinyML-based PHM applications where cloud computing could contribute to:

- Cloud-assisted model training and code generation for heterogeneous hardware devices.
- Data storage and integration for heterogeneous sensor data sources.
- Model inference computation offloading for latency, power and communication optimization.

The remainder of the paper is organized as follows: Section 2 describes the potential techniques for more efficient cooperation between cloud and TinyML models in PHM applications; Section 3 provides empirical experiment design and evaluation results of applications on hardware platforms; and Section 4 concludes the paper. Code available at: `https://github.com/dustinjoe/TinyML -with-Cloud-for-PHM`.

2. METHODOLOGY

In this section we delve into the details of the underlying methods used in our work. We first provide a general introduction to the current status of TinyML in Section 2.1. Then we propose three methods covering aspects of cloud computing that would help more efficient TinyML deployments in PHM applications in Section 2.2- 2.4.

2.1. TinyML: Machine Learning Inferencing on Tiny Microcontrollers

Microcontrollers (MCUs) are low-power computation devices with limited hardware resources. They are widely used in almost all cyber-physical systems including for PHM application. A typical microcontroller would have less than 500kb of storage and less than 100kb of RAM. In order to run a machine learning model on such a small-scale device, we need a light-weight model execution engine for the MCU as well as an optimized model representation. Even though in theory it is possible to also have model training updates on MCUs (Ren, Anicic, & Runkler, 2021), the limited amount of computation

power on MCUs can often make the model learning/updating process on MCUs infeasible. As a result, from a practical point of view, in this work, we focus on conducting machine learning inference rather than learning on MCU devices.

The development of TinyML enables a systematic application of relatively complex prediction models on MCUs (C. Banbury et al., 2021). For example, the work in (Crocioni, Pau, Delorme, & Gruosso, 2020) describes State of Health (SoH) for battery health monitoring and presents a comparison of different ML algorithms for estimating maximum releasable capacity of Li-Ion batteries. These efforts formulate the current tiny machine learning stack as consisting of sensor fusion, data collection, model training, model optimization, hardware-targeted model conversion and realistic hardware deployment (C. R. Banbury et al., 2020). Moreover, although there exists substantial heterogeneity in both software and hardware for different levels of model training and deployment, the machine learning model training are all based on general frameworks like Tensorflow/Keras and the deployment of these models on MCUs often involves a hardware-targeted model conversion and optimization procedure (David et al., 2020; Crocioni et al., 2021).

2.2. Cloud-Assisted Model Training/Deployment

Recently, there has been a growing trend towards minimizing manual code development (Broll & Whitaker, 2017) but rather have users utilize simple logic or drag-and-drop functional blocks to realize the desired features. To design a cyber-physical system with smarter decision making, a development platform purporting to minimizing coding efforts should be able to combine IoT sensing endpoints with hardware-targeted embedded machine learning model deployment. Furthermore, this platform should be able to automatically generate an optimized model that provides optimal on-device inference performance on custom hardware platforms. PHM applications are typical cyber-physical system applications that would involve a holistic workflow that requires both data collection and data processing. As a result, we can expect more end-to-end code generation and deployment workflows for PHM applications involving embedded machine learning.

With more sensors connected in PHM application, an integrated cloud platform would be critical for data integration as well as infrastructure management (Vuppalapati, Ilapakurti, Chillara, Kedari, & Mamidi, 2020). There have been some pioneering end-to-end solutions that have shown the prototype of these potential scenarios. Two examples are SensiML (Chris Knorowski, 2021) and Edge Impulse (Louis Moreau, Mihajlo Raljic, 2021). They both have integrated TinyML development workflows to enhance the model training and deployment on MCU or other edge devices. This kind of public cloud service helps lower the technical barrier to realize the workflows for edge model applications. It incorporates functions of data

acquisition, processing, model training&testing and actual device deployment. It also generates C code for various type of devices including MCUs.

2.3. Integration with Database Storage

Time/Sensor	Sensor_1	......	Sensor_n
Step_1	data(1,1)	...	data(1,n)
....	...		...
Step_T	data(T,1)	...	data(T,n)

Figure 1. Sensor Data Time Series.

For health status diagnosis and prognosis, we must often deal with continuous incoming sensor data. As a result, the model would absorb time series data from sensors and use the data format for predictions as shown in Fig 1. In addition, MCUs can operate as gateways to enable protocol translation and edge-to-cloud internet connection. To cope with tremendous amount of heterogeneous types of data, different cloud database systems have been exploited (Eyada, Saber, El Genidy, & Amer, 2020). To seek high performance in tasks like data transformation, analytics computations and data visualization, etc., specific database systems have been actively under development (Amghar, Cherdal, & Mouline, 2018). With respect to the edge-to-cloud PHM continuum presented in this paper, databases that provide effective support for interaction with ML-based data mining techniques are applicable to our work. One example includes the INFN-CNAF computing center (Umberto Griffo, 2019) used to launch large-scale data mining tasks based on InfluxDB (Nasar & Kausar, 2019) towards a global predictive maintenance solution. Apart from time series data, we also must deal with non-structural data like images, audio or even video. This adds to the necessity of using cloud as an integrated storage solution.

2.4. Computation Offloading

Even though TinyML development has enabled the execution of light-weight prediction models on MCUs, the limited amount of hardware resources on the MCU side cannot handle larger volumes of data from heterogeneous data sources. To solve this issue, we need to combine the cloud servers with MCU devices to formulate an edge-cloud computing paradigm for more adaptive computation burden offloading (Satyanarayanan, 2017).

One significant concern of features from sensor data in PHM applications is that the input data dimensions have semantic structural relationships. The main motivation for data-driven prognostics is that in practice we may not be able to get accurate physics-based models to conduct reliable model-based prognostics. But how to fuse information from distributed data sources for system prognosis in a hybrid way is still a practical problem. By offering more flexible data manipulation, we argue that the combination of TinyML and cloud computing can play an important role in PHM applications.

As we have mentioned, in practice we must deal with various kinds of data for PHM applications. This leads to challenges from two inevitable aspects when we combine it with the high-level cloud computing when we are seeking a system-level performance evaluation. One is the presence of large volumes of data communication between the MCU edge nodes and the cloud server. The other is the interaction between the TinyML model and the cloud-based ML model. These two aspects may manifest in various forms in practical applications but we propose a more general framework to solve these issues.

Based on the temporal dependency of the predictor input data, we classify ML-driven PHM applications into three types:

- No Temporal: e.g, comprising discrete data input like camera images.
- Continuous: e.g., those comprising continuous streams from all sensor nodes. Only new coming data is updated for storage and prediction.
- Discontinuous: e.g., those that do not need monitoring at all times. All sensor data in the past window length of time, however, needs to be fetched.

For the first setting without temporal dependencies, the TinyML predictor serves as an early warning as well as a data selector. It uses light-weight models to judge whether it is necessary to send the current input to the cloud for further analysis. If it is not necessary then the communication bandwidth for data transmission would be saved. This working mode would be meaningful for data with relatively large granularity.

Both continuous and discontinuous modes put emphasis on time series data, but the continuous mode conducts data analysis continuously and while the latter conducts analysis periodically or based on other monitoring policies. Irrespective, the general computation offloading guideline is that TinyML can only conduct light-weight predictions using nearby sensor data, must interact with the cloud and let the cloud server conduct the more systematic analysis. The discontinuous mode, however, needs to send more data to the cloud server for one inference cycle.

We propose a resilient TinyML model application strategy to solve this issue. We dive into the multi-layer structure of neural network models. For the working mode with no temporal or continuous dependency, the data uploaded remains unchanged. But for the discontinuous mode, the data uploaded to the cloud would be the latent space representation of one intermediate layer of neural network model. With a designed network structure, the latent space representation can have much fewer dimensions than the original input. In this way, the first part of the TinyML prediction model serves as an input decoder that helps compress the transmitted data. We show an example
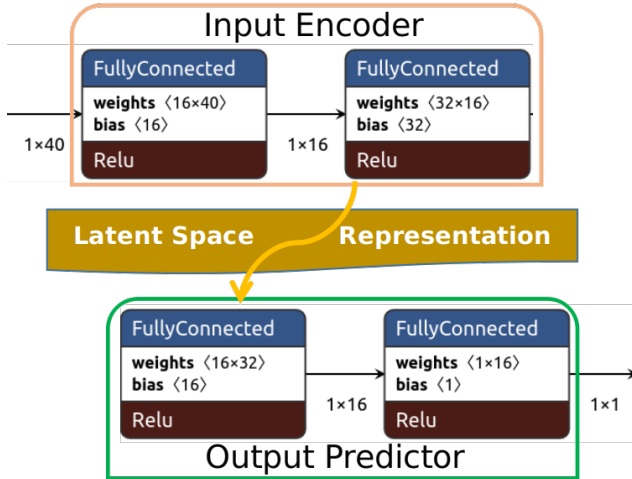
Figure 2. Resilient Application of TinyML Model.

in Fig 2. The input is first sent into an encoder and encoded as a latent representation. Then this latent representation is further utilized for predictions. This ideology could be utilized for both supervised and unsupervised learning models. For example, using an identity mapping auto-encoder is a classical unsupervised method for sparser data representations and also anomaly detection (Crocioni et al., 2021).
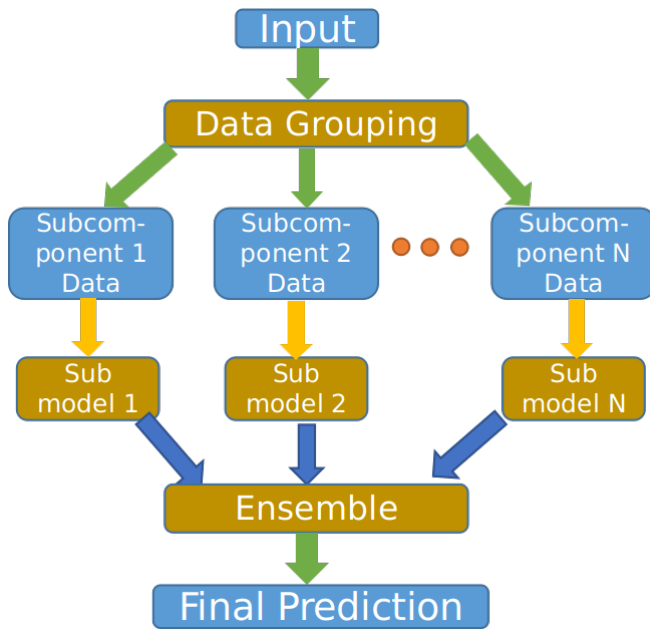

Figure 3. Ensemble System-level Modeling of TinyML Sub-models.

From the semantic structure point of view, TinyML models are focused on data from one sensor or a small group of sensors nearby. In this way, one model on MCU copes with data from one sub-component of the system. To get the overall system status prediction, we need to find a way to integrate results from sub-components together. In machine learning this can be defined as an ensemble as shown in Fig 5.

The strategy for an ensemble is flexible under different application settings. For the mode without temporal dependencies or continuous mode, this can be conducted as the process of fusing results from different sensors. This may involve the fusion of data as well as prediction results. For discontinuous mode, this could be the concatenation of compact latent space representations and then utilized together for more systematic analysis. This can be realized in a hierarchical manner. Lightweight models for sub-components are trained first and frozen. Then these pre-trained models can be combined together. This means the 'Ensemble' operation shown in Fig 5 can either be a relative simple operation like concatenation or some extra neural network layers going forward towards a final systematic result.

In summary, we discuss potential critical roles that TinyML can play together with cloud computing in data-driven PHM applications. We emphasize three critical functions of early predictive warning, dynamic offloading and data compression that TinyML can realize inc conjunction with a cloud setting.

3. EVALUATION

For demonstration purposes, we conduct two case studies to showcase the role of TinyML under different PHM working modes. The first study is about surface crack detection using camera images. The MCU takes pictures using onboard cameras and use a TinyML model to detect whether there is crack in the image. If one is detected, then it sends the image to cloud for further analysis. The second case study uses the widely-used C-MAPSS jet engine degradation dataset (Saxena & Goebel, 2008). We build light-weight models for sub-components of the engine and attempt to conduct both continuous and discontinuous prognostics using the TinyML techniques. It is worth pointing out that the C-MAPSS jet engine is not a perfect example for this distributed prognostics case study and we use it under an ideal assumption for demonstration purposes only.

3.1. Prototypical Hardware Deployment Validation

The practical question is whether these application models become too difficult to deploy yet need to have acceptable performance. To validate the practicality of deploying models on the cloud as well as on the edge, we provide a prototypical hardware deployment of these models on a server GPU and an edge MCU computation platform as follows:

- **Cloud Hardware:** The cloud inference deployment was conducted on a workstation (server) with a 4 Core/8 Thread Intel I7-7700HQ processor, 64GB RAM and a 2560-core NVIDIA P5000 Mobile Pascal GPU with 16GB GDDR5X VRAM.
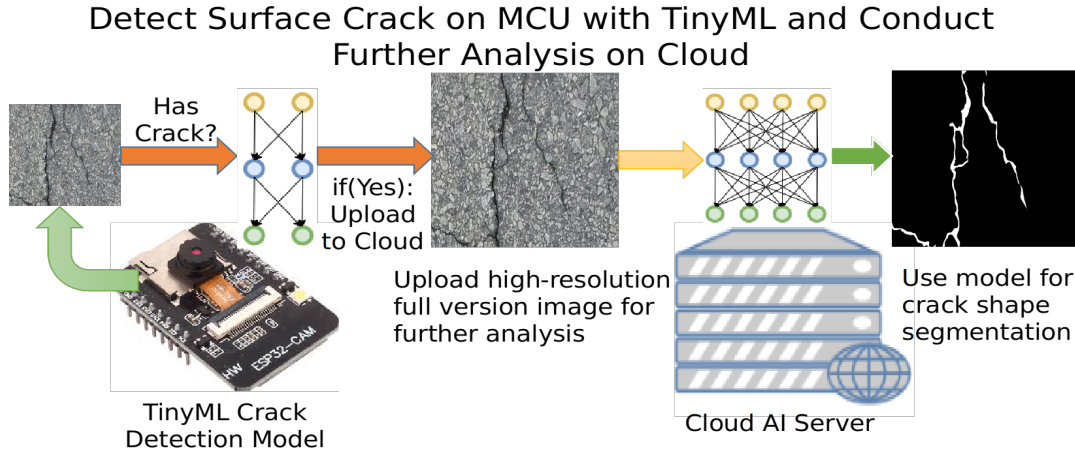
Figure 4. Crack analysis workflow of running binary crack detection on MCU using relative low-resolution images and further segmentation analysis on cloud using high-resolution images.

- **MCU Hardware:** The edge inference deployment was conducted on an ESP32-Cam Development Board with WiFi BT/BLE SoC module, a low-power dual-core 32-bit CPU with working frequencies of up to 240MHz, a built-in 520 KB SRAM with an external 4M PSRAM. It also supports OV2640 and OV7670 cameras with micro-sd card read/write options.

For the MCU hardware, we choose the ESP32 Wifi MCU family chip for our experiments. The ESP32 is a low-cost, low-power system on a chip series of microcontrollers integrated with Wi-Fi and Bluetooth capabilities. ESP32 uses a IoT-targeted architecture powered by a dual-core Tensilica Xtensa LX6 microprocessor, which is more specific than ARM-Cortex series processors. ESP32 development is also officially supported by the popular MCU development tool of Arduino IDE so that many existing packages can be used on ESP32 without any barrier. Compared to Arduino MCUs, ESP32 has more hardware resources and communication methods embedded so it has become a popular choice for state-of-the-art IoT project development.

As we are only demonstrating the applicability of TinyML deployment on the device, we use the USB cord to power the device. But it is worth pointing out, in practice the low power consumption of these MCUs enable the possibility of working in totally wireless mode using solar power (Priambodo & Nugroho, 2021).

3.2. Case 1: Surface Crack Analysis

3.2.1. Problem Description

Crack occurrence and propagation are critical factors that affect the reliability of solid structures. Correspondingly, crack analysis plays a vital role in health monitoring and inspection of structures like buildings/bridges/roads/equipment. Tradi-

tional inspection on cracks require heavy manual load, and image processing and computer techniques help formulate automatic analysis pipelines (Mohan & Poobal, 2018).

We consider a crack analysis pipeline consisting of two phases of detection and segmentation (Zou et al., 2018). Crack detection judges whether there exists cracks in the region of a given structural image (L. Zhang, Yang, Zhang, & Zhu, 2016). Given an input image with likely damage, crack segmentation further detects crack locations on images in a more detailed and accurate manner (X. Zhang, Rajan, & Story, 2019). For our study, the detection model training and evaluation is conducted using the Concrete Crack Images for Classification Dataset (Özgenel & Sorguç, 2018). The segmentation model training and evaluation is conducted using the Crack500 Dataset (X. Zhang et al., 2019).

3.2.2. System Model

Although past efforts have shows reliable analysis from static images, with the development of TinyML, we can imagine a more flexible deployment of crack detector such as on a MCU-assisted structure patrol robot and provide detection feedback as early warnings in a real-time manner and send images with potential cracks to the cloud for further analysis.

In order to realize the working pipeline proposed above, we need to allocate different computation burdens to suitable devices as shown in Fig 4. We propose the device workflow of running binary crack detection on MCU using relative low-resolution $(32, 32, 3)$ RGB images and segmentation analysis on cloud using high-resolution $(224, 224, 3)$ RGB images.

3.2.3. Results

These years' development of deep learning has shown the success of convolutional neural networks (Özgenel & Sorguç,

	Model Size	Precision	Input Type	Output Type	Latency	Performance
MCU Detection	584.9 kb	INT8	(32,32,3) RGB	Yes/No	479 ms	Accuracy: 99.5%
Cloud Detection	22.4 mb	Float32	(224,224,3) RGB	Yes/No	2.4 ms	Accuracy: 99.7%
Cloud Segmentation	40.8 mb	Float32	(224.224.3) RGB	Segmentation Mask	7.3 ms	mIOU: 0.5075

Table 1. Crack analysis performance on devices. MCU side runs a quantized model for this binary detection inference task. It gains good classification accuracy performance with only about $1/40$ model size. In contrast, segmentation has high computation burden and can only be deployed on the cloud.

2018). Both the cloud and MCU side models are used for computer vision tasks and we make use of mature convolutional neural network(CNN) architectures for these tasks.

On the MCU side, we make use of the Edge Impulse cloud platform (Louis Moreau, Mihajlo Raljic, 2021) to train a crack detection model. This model training is using transfer learning based on a pre-trained Mobilenet (Howard et al., 2017) for computer vision tasks. The training process takes less than 10 minutes with 6000 training images and 2000 testing images with a resolution of $32 * 32$. The cloud platform also generated C code for MCU deployments.[1]

On the cloud side, we use the Nvidia GPU for model training. The segmentation model is based on the U-Net (Ronneberger, Fischer, & Brox, 2015) architecture. It takes in high-resolution surface images and returns segmentation masks to show the exact location and shape of cracks in the region.

We summarize experimental results in Table 1. We can see that for this binary detection task, the TinyML model on the MCU achieves as good classification accuracy as the cloud model with only about $1/40^{th}$ of model size. One thing worth noticing is that the model on the MCU side is an optimized compressed version with 8-bit integer weight/activation quantization (Han, Mao, & Dally, 2015). Compared to full-precision 32-bit floating point representations, 8-bit representations save significantly on model size. The segmentation models need to compute and output the actual crack region. The computation burden and file read/write size is much higher than the binary detection. MCUs have neither enough computation power nor storage space for further segmentation analysis. Thus, it is necessary to put the segmentation model on the cloud side. The evaluation metric of this segmentation task is called Mean Intersection-Over-Union (mIOU), which refers to the average prediction bounding box overlapping for classes of objects (X. Zhang et al., 2019). Our implementation is based on the work presented in (Chen, Liu, & Chen, 2019) and has achieved normal performance on this segmentation task.

3.3. Case 2: CMAPSS Jet Engine Prognostics

3.3.1. Problem Description

The C-MAPSS dataset (Saxena & Goebel, 2008) is a dataset for data-driven remaining useful life (RUL) prediction gener-

ated from detailed simulations of jet turbofan engines. The data trace for an engine starts from a degrading time point and ends (RUL=0) at the end of the running cycle. Apart from time label, there are 24 features for each data point as shown in Table 2. The first three are the three operational settings that have a global impact on engine performance.The remaining ones represent the 21 sensor values from different sub-components of the engine system.

No.	Parameter	Detail	Group
1	C1	Control input 1	1
2	C2	Control input 2	1
3	C3	Control input 3	1
4	T2	Total temperature at fan inlet	2
5	T24	Total temperature at LPC outlet	2
6	T30	Total temperature at HPC outlet	3
7	T50	Total temperature at LPT outlet	4
8	P2	Pressure at fan inlet	2
9	P15	Total pressure in bypass-duct	2
10	P30	Total pressure at HPC outlet	3
11	Nf	Physical fan speed	2
12	Nc	Physical core speed	4
13	epr	Engine pressure ratio (P50/P2)	1
14	Ps30	Static pressure at HPC outlet	3
15	phi	Ratio of fuel flow to Ps30	3
16	NRf	Corrected fan speed	2
17	NRc	Corrected core speed	4
18	BPR	Bypass Ratio	2
19	farB	Burner fuel-air ratio	4
20	htBleed	Bleed Enthalpy	1
21	Nf_dmd	Demanded fan speed	2
22	PCNfR_dmd	Demanded corrected fan speed	2
23	W31	HPT coolant bleed	4
24	W32	LPT coolant bleed	4

Table 2. Sensor feature grouping according to their semantic structural contexts. Different groups refer to different sub-components of the system.

For demonstration purpose, we only choose the first set $FD001$ for our experiments here. Data preparation steps are conducted on this dataset including all zero value feature removal and normalization into the 0.0-1.0 value range using MinMax (Patro & Sahu, 2015). The remaining 17 features can be divided into 4 groups based on their semantic structural locations.

1. Global: 1, 2, 20
2. Fan+Splitter: 11, 16, 5, 9, 18
3. HPC+Fuel: 6, 10, 14, 15
4. Burner+HPT+LPT+CoreNozzle: 7, 23, 24, 12, 17

We regard sensors of these different groups as spatially separated and have MCUs deployed in groups of 2, 3, 4 and explore

---

[1]Project publicly available on EdgeImpulse: `https://studio.edgeimpulse.com/public/32815/latest`

the cooperation of tiny prognostics models on MCUs and the holistic prediction model on the cloud. It is worth pointing out that this distributed prognostics system setting is based on several ideal assumptions. First of all, we regard these sensors as separable into distributed groups and can be connected to MCUs separately. In addition, we do not consider the extreme condition (like high temperature or pressure in the engine) working applicability of MCUs. Moreover, the experiments are conducted on stable Wifi connections. Overall, we regard this as a use case for demonstrating a scenario that aims to explore potential settings for distributed prognostics from different sensors (installed on sub-components) in a sensor network. We are using this dataset partly due to the fact that we failed to find a better dataset for flexible demonstration of use cases under diverse data division settings.

3.3.2. System Model

Based on the overall problem discussed above, we build models on both MCUs and cloud servers to conduct prognostics. From the functionality point of view, TinyML on-device prognostics provides the early warning from the sub-component level and the cloud prognostics provides the more accurate system-level analysis. Here we use TFLite-Micro (Tensorflow Development Team, 2021) from Tensorflow team to transform machine learning models and execute models on MCUs with C-based interpreters (Simone, 2020).

For this RUL value regression problem, we use a relatively simple network architecture on the MCU side. We use a three-layer multilayer perceptron (Goodfellow, Bengio, & Courville, 2016) with $16, 32, 16$ neurons in intermediate layers. This simple model structure is also shown in Fig 2. This specific network refers to the model architecture of MCU2, which absorbs data from Group2. The input of the model is flattened into one dimension and sent into the model for forward inference computations. The input of dimension 40 is first sent into an encoder model and encoded as a latent representation of dimension 32. Then this latent representation is further utilized for remaining useful life regression prediction and outputs one floating point value as the final result.

One thing worth mentioning is that the input data we are dealing with belong to time series values so it would be a more optimal choice to use network models with temporal considerations like LSTM (Hochreiter & Schmidhuber, 1997) in recurrent neural networks. Unfortunately, these recurrent neural networks have not been supported as kernel operations in current TFLite-Micro (Tensorflow Development Team, 2021) yet.

For the continuous working mode, all sensor data updates are sent to the cloud server. We build a standard MySQL server as the cloud database to store these incoming time series data. On the cloud side with full machine learning framework support, a more standard LSTM network is built for more accurate

prognostics. Here the LSTM model has two layers with 100 and 50 units. Moreover, the output of the second LSTM layer is connected to one fully-connected neuron to output one final prediction value.
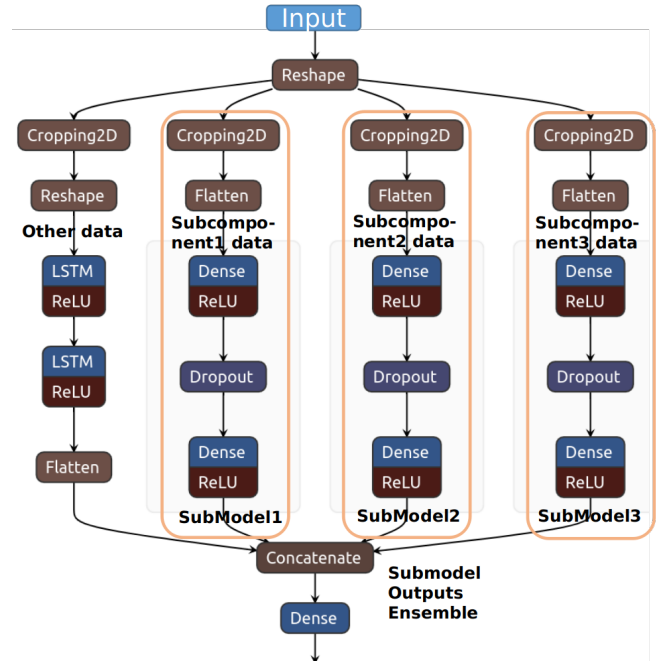

Figure 5. Model ensemble paradigm for the incontinuous working mode. Here we show the concatenation of latent space representations from three MCU submodels.

For the discontinuous working mode, we need to fetch a batch of data from a set window length of time to conduct prognostics tasks. We propose a submodel ensemble paradigm as shown in Fig 5. We first train MCU submodels for sub-components with different groups of sensor data. We then concatenate outputs of these tiny models but freeze parameter values in these tiny models. The global impact features are only used in the high-level holistic model. In this way, we retrain this holistic model for an overall prediction. One key point in the submodel is that we are only concatenating the outputs of the first half of the 'Input Encoder' of the Fig 2. In this way, we are only transmitting the latent space representation values with fewer dimensions than the model input and the communication bandwidth can be saved.

3.3.3. Results

Even though we are focused on the tiny machine learning model deployments on MCUs, current 32-bit MCUs like ESP32 can actually be used in more flexible ways. For this use case, the ESP32 MCU serves as a simple but multi-functional server. Fig 6 shows an example of such a simple server running on MCU. It can support TinyML prediction along with data visualizations. It stores sensor data for a past certain length of time. When new data comes in, it would post these

	Model Size	Precision	#Params	Latency /μs	Regression MAE	Data Transmission
MCU1	57.9 kb	Float32	1905	638	26.29	fp32*6
MCU2	54.0 kb	Float32	1745	588	27.47	fp32*5
MCU3	58.0 kb	Float32	1905	647	25.15	fp32*6
Cloud	1005.6 kb	Float32	35251	94.2	24.02	fp32*17

Table 3. RUL Predictions Under Continuous Mode

|	Model Size /kb	Model Precision	#Params	Latency / μs	Regression MAE	Data Transmission Uncompressed	Data Transmission Compressed	Bandwidth Save
MCU1	40.5+20.3	Float32	1360+545	531+152	26.29	fp32*50	fp32*32	36%
MCU2	36.5+20.3	Float32	1200+545	487+155	27.47	fp32*40	fp32*32	20%
MCU3	39.6+20.3	Float32	1360+545	522+161	25.15	fp32*50	fp32*32	36%
Cloud	168.4	Float32	9137	104	24.7	fp32*17*10	fp32*(96+30)	25.90%

Table 4. RUL Predictions Under Incontinuous Mode
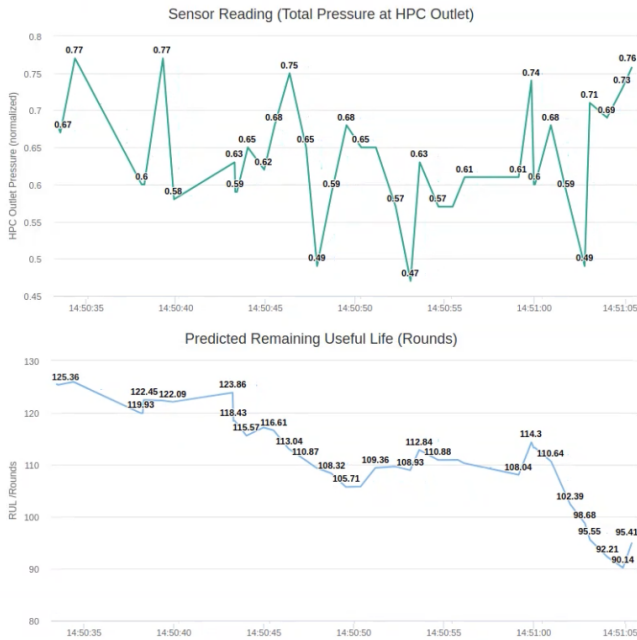
Figure 6. A tiny prognostics server deployed on MCU. It can support TinyML prediction along with data visualizations.

data to the cloud server to make global RUL prediction using a more holistic model. But on the ESP32 itself it runs a lightweight RUL prognostics model for on-device inference, so if it reaches a threshold it should actively send alarm to the client user. Making use of these tiny MCU prognostics servers, we gather data and run models with these data.

We show experiment results from the continuous mode in Table 3. For value regressions, we cannot use quantized models to save model size. But still these tiny prediction models can run on low-power MCUs with low-latency. The metric of mean absolute deviation (MAE) is used for evaluation of the model performance.

Similarly, we show experiment results from the discontinuous mode in Table 4. As we have mentioned, to get access to the intermediate latent space representation, we divide the original predictor into the input encoder and the output predictor. The total number of model parameters remain the same. But some storage cost has been induced to store more model structural information. With the latent space representation, the data transmission size can be greatly reduced. Again, we get an overall better system-level prediction with the information from all submodels and the global features.

4. CONCLUSION

Large scales and big data volumes in practice call for more resilient data-driven models in health management and prognostics scenarios. Development of current machine learning, especially deep learning enables the execution of prediction models on different levels of computation devices. This paper serves as a vision paper for the novel field of TinyML opportunities on MCU devices in PHM application scenarios. We not only emphasize the potential of TinyML but also the significance of combining it with high-level cloud computing from a more systematic point of view. We explore the question of how cloud platform can help build efficient TinyML models. We show the significance of inducing cloud resources for data storage and integration. Furthermore, we investigate the possible application patterns for more adaptive computation burden offloading from the edge MCU to the cloud. Our discussions and experimental implementations on two case studies cover the most typical settings in PHM applications. These general settings would help formalize the system resilience testing in more scenarios.

Our future work is focused on the following three aspects. First, our current investigation is based on the Wifi network as the communication method. Practical PHM application scenarios may be more complex for such a universal wireless connection. We may need to further investigate the impact of unreliable data communication on TinyML deployments. Secondly, we demonstrate the possibility of using machine

learning model for data analysis and use the latent space representation as the data compression to save bandwidth. But there are other methods that are also worth exploring like PCA and clustering. In addition, due to lack of support for RNNs in current TFLite-Micro (Tensorflow Development Team, 2021), we have not achieved ideal performances on time series data. This also limited us from exploring unsupervised models like LSTM Autoencoders (Crocioni et al., 2021). Thirdly, even though our system-level ensemble framework has shown promising performance on test settings, the strategy of model ensemble is still under manual configuration. Therefore, we need to find out ways to search for more flexible model ensemble strategies.

ACKNOWLEDGMENT

REFERENCES

Amghar, S., Cherdal, S., & Mouline, S. (2018). Which nosql database for iot applications? In *2018 international conference on selected topics in mobile and wireless networking (mownet)* (pp. 131–137).

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

Banbury, C. R., Reddi, V. J., Lam, M., Fu, W., Fazel, A., Holleman, J., . . . others (2020). Benchmarking tinyml systems: Challenges and direction. *arXiv preprint arXiv:2003.04821*.

Broll, B., & Whitaker, J. (2017). Deepforge: An open source, collaborative environment for reproducible deep learning.

Chen, J., Liu, G., & Chen, X. (2019). Road crack image segmentation using global context u-net. In *Proceedings of the 2019 3rd international conference on computer science and artificial intelligence* (pp. 181–185).

Chris Knorowski. (2021). *Building a TinyML Application with TF Micro and SensiML.* {https://blog.tensorflow.org/2021/ 05/building-tinyml-application-with -tf-micro-and-sensiml.html}. ([Online; accessed 19-June-2021])

Crocioni, G., Gruosso, G., Pau, D., Denaro, D., Zambrano, L., & Di Giore, G. (2021). Characterization of neural networks automatically mapped on automotive-grade microcontrollers. *arXiv preprint arXiv: 2103.00201.*

Crocioni, G., Pau, D., Delorme, J.-M., & Gruosso, G. (2020). Li-ion batteries parameter estimation with tiny neural networks embedded on intelligent iot microcontrollers. *IEEE Access*, *8*, 122135–122146.

David, R., Duke, J., Jain, A., Reddi, V. J., Jeffries, N., Li, J., . . . others (2020). Tensorflow lite micro: Embedded machine learning on tinyml systems. *arXiv preprint arXiv:2010.08678.*

Eyada, M. M., Saber, W., El Genidy, M. M., & Amer, F. (2020). Performance evaluation of iot data management using mongodb versus mysql databases in different cloud environments. *IEEE Access*, *8*, 110656–110668.

Farinholt, K. M., Chaudhry, A., Kim, M., Thompson, E., Hipwell, N., Meekins, R., . . . Polter, S. (2018). Developing health management strategies using power constrained hardware. In *Phm society conference* (Vol. 10).

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

Grosvenor, R. I., & Prickett, P. W. (2011). A discussion of the prognostics and health management aspects of embedded condition monitoring system. In *Annual conference of the phm society* (Vol. 3).

Han, S., Mao, H., & Dally, W. J. (2015). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149.*

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., . . . Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861.*

Louis Moreau, Mihajlo Raljic. (2021). *Enhancing Health and Safety in Industrial Environments with Embedded Machine Learning.* {https://www.edgeimpulse.com/ blog/enhancing-health-and-safety -in-industrial-environments-with -embedded-machine-learning}. ([Online; accessed 19-June-2021])

Mohan, A., & Poobal, S. (2018). Crack detection using image processing: A critical review and analysis. *Alexandria Engineering Journal*, *57*(2), 787–798.

Nasar, M., & Kausar, M. A. (2019). Suitability of influxdb database for iot applications. *International Journal of Innovative Technology and Exploring Engineering*, *8*(10), 1850–1857.

Özgenel, Ç. F., & Sorguç, A. G. (2018). Performance comparison of pretrained convolutional neural networks on crack detection in buildings. In *Isarc. proceedings of the international symposium on automation and robotics in construction* (Vol. 35, pp. 1–8).

Patro, S., & Sahu, K. K. (2015). Normalization: A preprocessing stage. *arXiv preprint arXiv:1503.06462*.

Priambodo, A., & Nugroho, A. (2021). Design & implementation of solar powered automatic weather station based on esp32 and gprs module. In *Journal of physics: Conference series* (Vol. 1737, p. 012009).

Ren, H., Anicic, D., & Runkler, T. (2021). Tinyol: Tinyml with online-learning on microcontrollers. *arXiv preprint arXiv:2103.08295*.

Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. In *International conference on medical image computing and computer-assisted intervention* (pp. 234–241).

Satyanarayanan, M. (2017). The emergence of edge computing. *Computer*, *50*(1), 30–39.

Saxena, A., & Goebel, K. (2008). C-mapss data set. *NASA Ames Prognostics Data Repository*.

Simone. (2020). *EloquentTinyML Arduino Library.* `{https://github.com/eloquentarduino/EloquentTinyML}`. ([Online; accessed 10-July-2021])

Song, Q., Luan, F., Shi, Z., Li, T., & Wang, M. (2020). Design of turbidity remote monitoring system based on fx-11a optical fiber sensor. In *2020 prognostics and health management conference (phm-besançon)* (pp. 291–294).

Tensorflow Development Team. (2021). *TFLite Micro Kernel Ops.* `{https://github.com/tensorflow/tflite-micro/tree/main/tensorflow/lite/micro/kernels}`. ([Online; accessed 10-July-2021])

Umberto Griffo. (2019). *Recurrent Neural Networks for Predictive Maintenance.* `{https://github.com/umbertogriffo/Predictive-Maintenance-using-LSTM}`. ([Online; accessed 19-May-2020])

Vuppalapati, C., Ilapakurti, A., Chillara, K., Kedari, S., & Mamidi, V. (2020). Automating tiny ml intelligent sensors devops using microsoft azure. In *2020 ieee international conference on big data (big data)* (pp. 2375–2384).

Warden, P., & Situnayake, D. (2019). *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers.* ” O'Reilly Media, Inc.”.

Zhang, L., Yang, F., Zhang, Y. D., & Zhu, Y. J. (2016). Road crack detection using deep convolutional neural network. In *2016 ieee international conference on image processing (icip)* (pp. 3708–3712).

Zhang, X., Rajan, D., & Story, B. (2019). Concrete crack detection using context-aware deep semantic segmentation network. *Computer-Aided Civil and Infrastructure Engineering*, *34*(11), 951–971.

Zou, Q., Zhang, Z., Li, Q., Qi, X., Wang, Q., & Wang, S. (2018). Deepcrack: Learning hierarchical convolutional features for crack detection. *IEEE Transactions on Image Processing*, *28*(3), 1498–1512.