A Primer for tinyML Predictive Maintenance: Input and Model Optimisation

Emil Njor, Jan Madsen, Xenofon Fafoutis

▶ To cite this version:

Emil Njor, Jan Madsen, Xenofon Fafoutis. A Primer for tinyML Predictive Maintenance: Input and Model Optimisation. 18th IFIP International Conference on Artificial Intelligence Applications and Innovations (AIAI), Jun 2022, Hersonissos, Greece. pp.67-78, 10.1007/978-3-031-08337-2_6 . hal-04668653

HAL Id: hal-04668653

https://inria.hal.science/hal-04668653v1

Submitted on 7 Aug 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

This document is the original author manuscript of a paper submitted to an IFIP conference proceedings or other IFIP publication by Springer Nature. As such, there may be some differences in the official published version of the paper. Such differences, if any, are usually due to reformatting during preparation for publication or minor corrections made by the author(s) during final proofreading of the publication manuscript.

A Primer for tinyML Predictive Maintenance: Input and Model Optimisation

Emil Njor[0000-0003-0219-1624], Jan Madsen[0000-0002-5098-8454], and Xenofon Fafoutis[0000-0002-9871-0013]

DTU Compute, Technical University of Denmark, Denmark
{emjn,jama,xefa}@dtu.dk

Abstract. In this paper, we investigate techniques used to optimise tinyML based Predictive Maintenance (PdM). We first describe PdM and tinyML and how they can provide an alternative to cloud-based PdM. We present the background behind deploying PdM using tinyML, including commonly used libraries, hardware, datasets and models. Furthermore, we show known techniques for optimising tinyML models. We argue that an optimisation of the entire tinyML pipeline, not just the actual models, is required to deploy tinyML based PdM in an industrial setting. To provide an example, we create a tinyML model and provide early results of optimising the input given to the model.

Keywords: tinyML · Predictive Maintenance · Optimisation · Embedded Machine learning · Resource-Constrained Systems

1 Introduction

Predictive Maintenance (PdM) is a promising maintenance paradigm where models are used to predict equipment failure. PdM is expected to replace reactive maintenance and preventive maintenance [23]. In reactive maintenance, equipment is repaired after a failure which, e.g. in factories can lead to expensive production downtime. In preventive maintenance, equipment is replaced according to a predefined schedule. This can be wasteful as perfectly working equipment might be replaced. It also provides no guarantee that failures do not occur before maintenance. Assuming a perfect model, PdM can predict a failure before it occurs, and maintenance can be planned and conducted in advance to avoid the failure [23].

There are three general approaches to implementing PdM systems. The first is a knowledge-based approach, which uses e.g. rules or physical models to predict failures. The second and third are traditional Machine Learning (ML) and Deep Learning approaches respectively [23]. The output of PdM models generally come in three different forms. One is a binary prediction, where the model outputs whether there is an impending equipment failure. A second form is anomalous behaviour detection. In this form, the model flags equipment behaviour as normal or anomalous. The third form of predictions is a Remaining Useful Life (RUL) prediction, where the RUL of some equipment is estimated [23].

At the moment, most PdM systems are deployed either in the cloud or on powerful computers. The data used by such models, however, are often generated by small sensor devices. Therefore, using current approaches, data has to be collected and sent over a network for processing. This has a couple of drawbacks e.g: (i) Security and privacy of data can be compromised when sent over a network. (ii) Network communication induces a non-zero and often unpredictable latency, which can be intolerable in some use cases. (iii) The system will be less reliable as it relies on a working network connection and cloud. This is especially a problem for systems deployed in rural areas or at sea. (iv) Using network modules on sensor devices requires a significant amount of energy. This is especially a problem for battery-driven sensor devices. The alternative to sending data over a network for processing is to process data directly on the sensor device. This has been popularized as the concept of tinyML [8].

According to the tinyML Foundation [8], tinyML is broadly defined as: *"A fast growing field of machine learning technologies and applications including hardware, algorithms and software capable of performing on-device sensor data analytics at extremely low power, typically in the mW range and below, and hence enabling a variety of always-on use-cases and targeting battery operated devices."* tinyML thus provides a solution to the drawbacks of sending data over a network to be processed. tinyML has its own challenges, however, and it is therefore heavily dependent on the use case whether a tinyML or a cloud solution is the better choice. The following are some challenges of tinyML: (i) The microcontrollers embedded in sensor devices have few computational resources, so inference of models will take significantly longer than in the cloud or on desktops. (ii) Microcontrollers also have a limited amount of memory, so the size of the models deployed with tinyML will have to be small. (iii) Microcontrollers often have little to no operating system, which means that tinyML can not rely on standard operating system features such as dynamic memory allocation. (iv) There are a wide variety of microcontrollers on the market, and the hardware and their software tools are heterogeneous.

The field of tinyML is still in its infancy, and work has to be put into making it ready for industrial adaptation. The contribution of this work is twofold. We first provide the background of how to apply tinyML based PdM in a tutorial style, and a short (yet – to the best of our knowledge – comprehensive) survey of known optimisation techniques for tinyML models. Secondly, we argue that it is important to optimise the entire PdM pipeline and not just the tinyML models to mature tinyML based PdM As an example, we create a tinyML based PdM model and show that by optimising the input to the model we can further reduce the compute and memory requirements for running inference of the model. We show the tinyML optimisation pipeline that we argue for in Figure 1.

2 Related Work

While the field of tinyML is still in its infancy, it has received much attention in recent years. Several surveys have been published on the topic. The surveys,

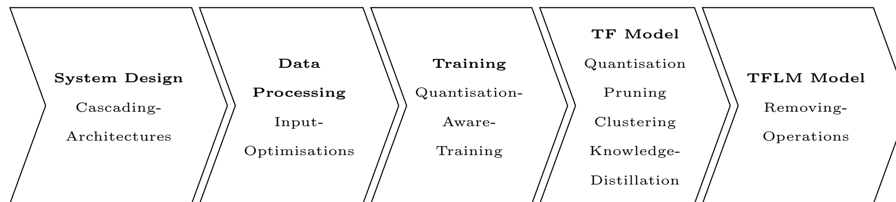

Fig. 1. The tinyML optimisation pipeline. Pipeline stages in bold followed by optimisations that are possible at the respective stages. More information about each optimisation is given in Section 4 and 5. TF and TFLM are abbriviations to TensorFlow and TensorFlow Lite Micro respectivly. See Section 3 for more information about TF and TFLM.

however, mostly investigate tinyML for any ML application, and not specifically PdM. In doing so, they tend to focus on common supervised learning techniques and cases. These techniques and cases differ from PdM in especially two areas. Firstly, as we shall discuss later, PdM datasets are often imbalanced or unlabeled. Secondly, PdM is a bit unique as late failure predictions in many cases should be penalised more heavily than early failure predictions. One tinyML survey is focused on anomaly detection [28]. This survey is focused on the types of anomaly detection systems and the techniques that are used to reduce the size of the ML models. It does not go into depth with the applications, libraries, hardware or datasets that are typically used for tinyML based anomaly detection. In this paper, we touch on these specifically for anomaly detection for PdM.

A survey by R. Sanchez-Iborra and A. Skarmeta [26] investigates general tinyML benefits, challenges, applications, libraries and models. The survey concludes with a case study, for which a decision tree model is deemed the best tinyML model choice [26]. The data for the case study is synthetically constructed, however, and is arguably constructed using decision tree-like logic, so the great performance of the decision tree is not surprising It does not comment on common datasets or hardware for tinyML. It also does not go into depth about optimising tinyML models. Another survey by L. Dutta et al. [5] describes the benefits of tinyML, and compares tinyML to other approaches to processing sensor data. It also touches on the hardware-software co-design, optimisation techniques, libraries and tools, recent advances and the role of the industry in tinyML research. A few example datasets and tinyML results on these datasets are also presented [5]. Lastly, a survey from P. Ray [25] describes hardware, libraries, optimisations and use cases. It also includes the authors' ideas for a future roadmap of tinyML. This is a quite comprehensive survey, but does not comment on common tinyML datasets.

A book on tinyML using Tensorflow Lite Micro [32] acts as a tutorial for deploying several types of ML applications on ARM-based microcontrollers. The book contains a comprehensive overview of the background of tinyML and Optimisation techniques, however, it does not investigate PdM or anomaly detection.

Apart from investigating tinyML, this paper also tackles optimisations of the input to tinyML models. To our knowledge, this has not yet been investigated in the domain of tinyML based PdM. It has, however, been applied in other fields. For example, a publication by X. Fafoutis et al. describes how a cloud-based ML system can be optimised by conducting feature extraction directly on a sensor device. This is shown to reduce the required compute and networking by several orders of magnitude [7]. Another paper by A. Khan et al. found that sampling rates used in the literature are up to 57% higher than needed [14].

3 Background of Applying tinyML

In this section, we present the background knowledge required to apply tinyML, with a focus on tinyML for PdM. The section can be regarded as a mini-survey of the libraries, hardware, datasets and models which are used in tinyML and PdM research; and can serve as a tutorial for individuals that enter the field.

Libraries. Several software libraries have been introduced to ease the development of tinyML applications. The most popular is arguably the open-source TensorFlow Lite Micro (TFLM) library, whose primary contributor is Google [4]. We base this on the tinyML book being focused on this library [32]. Furthermore, the online learning platform "edX" has a course on tinyML that focuses on the library [6].

The TFLM library is split into two parts. One part is a converter which converts TensorFlow (TF) models to the TFLM model format. TF is an open-source framework for creating, training and inferring Neural Network (NN) models also spearheaded by Google. The second part is an interpreter written in C++, which runs on a microcontroller and interprets the TFLM model. As the TFLM interpreter is designed for microcontrollers, it has low compute and memory requirements and does not rely on an operating system. TFLM does not support training and only supports a subset of the operations supported by TF. The library is also able to take advantage of efficient implementations of common neural network functions on ARM-based microcontrollers described in [17].

Alternatives to using TFLM include Microsoft's Embedded Learning Library, ARM-NN, sklearnporter and µTensor [26]. Most of these libraries are made for NNs as opposed to traditional ML, suggesting that the current focus of tinyML is leaning heavily towards NNs.

Hardware. This section describes some of the hardware that can be used for doing PdM inference using tinyML. We focus our attention on what we see as the current standard systems for deploying tinyML - that is ARM-based microcontrollers, but also comment on alternative hardware platforms. ARM-based microcontrollers can be split into two sub-groups - those building on the ARM Cortex-M platform and those building on the ARM Cortex-A platform. The ARM Cortex-M platform is the most energy-efficient of the two, while the ARM

Cortex-A platform provides superior performance [1]. The Cortex-A platform is used in devices like smartphones and the Raspberry Pi, whereas the Cortex-M platform is mainly used in embedded devices such as sensors.

TFLM as discussed in Section 3 is made for the Cortex-M platform, and confirmed supported for a number of devices listed on the TFLM webpage [30]. One of the supported devices is the Arduino Nano 33 BLE Sense. The device has an ARM Cortex-M4 processor, 256 KB of SRAM and 1 MB of flash memory. It furthermore has a wide array of internal sensors, which allows the device to be used as a prototype for several applications.

The literature proposes many alternatives to using ARM-based microcontrollers. One approach is to use the open-source processor platform PULP to speed up tinyML inference [9]. A paper shows that this platform can complete inference on a CIFAR-10 network in up to 30x fewer clock cycles than the current state of the art ARM-based microcontrollers [9]. Other publications propose completely new hardware such as BinarEye, which is a processor optimized for efficient processing of Convolutional Neural Networks (CNNs) [20]. Some papers even propose alternative ways to encode data in hardware to improve the processing energy efficiency [31].

Datasets. In order to implement tinyML based PdM appropriate data is needed. It is unfortunately notoriously difficult to find good datasets for predictive maintenance. One reason is that failures can lead to financial and reputational loss, especially in industry, and we often go to great lengths to avoid failures. Paradoxically it is also the goal of PdM to avoid failures. Even when a failure occurs, the data about the failure is often not released publicly.

We have identified three suitable datasets for tinyML based PdM. The first is the ToyADMOS dataset, which contains audio recordings of toys in normal and anomalous operating conditions [15]. A subset of the ToyADMOS dataset is used in the MLPerf Tiny benchmark, which to our knowledge is the only current benchmark targeting tinyML [2]. Similarly, we have the MIMII dataset that contains audio recordings of industrial machines in normal and anomalous operating conditions [22]. The third dataset is the Turbofan Engine Degradation dataset which contains sensor readings of simulated turbofan engines as they degrade towards failure [27].

An issue in many PdM datasets is the imbalance of observations. By their nature, normal operating conditions are more frequent than anomalous, and thus the data includes a majority of normal observations. The literature proposes to use generative models or transfer learning to solve these problems [23].

Models. The right model to choose for tinyML based PdM depends on the hardware and the data that is available for the PdM application. If the hardware capabilities are extremely limited, such as in the ATMega328P Microcontroller, which has only 2 KB RAM, then simple traditional ML methods, such as decision trees could be the right choice. For example, while the size of the TFLM interpreter used for NNs varies by the model that it needs to interpret, even the

smallest example in the TFLM paper takes up 1.3 KB of RAM [4]. In a 2 KB RAM microcontroller, this would not leave much room for the model, data and remaining logic. More specialised models can also be considered in this case, e.g the bonsai model, which is derived from decision trees [16]. In other microcontrollers such as the Arduino Nano 33 BLE Sense with 256 KB of RAM, NNs, especially for image processing, might be the better choice.

If the data is labelled, either with impending failure labels or RUL labels, then supervised learning approaches are likely the best choice. There are many supervised learning approaches [23], but Decision Trees, Support Vector Machines (SVMs), Artificial Neural Networks (ANNs) or CNNs can in our opinion all be good choices. The actual decision depends on other factors. CNNs are NNs that contain convolutional layers. Convolutional layers train a filter to pass over a tensor, usually an image, to extract features that help the classification/regression.

If the data is unlabeled then we need to turn to unsupervised learning. When this is the case we are often trying to do PdM anomaly detection. There are a few ML models that are suitable for making anomaly detection. Two of the most popular models are k-Nearest Neighbor (KNN) and autoencoders [23]. A KNN model is a traditional ML model which clusters observations based on features derived from the observations. The idea is that an anomalous sample will diverge from the cluster(s) of normal observations and that it can thereby be stamped as an anomaly. An autoencoder model is a deep learning model, which we train to compress and decompress normal observations. We also say that the autoencoder is "reconstructing" its input. The idea for this model is that the autoencoder will learn to reconstruct normal observations, but that it will struggle to reconstruct anomalous observations. Just as with other NNs we can introduce convolutional layers to autoencoders to improve their capabilities in image processing. In this case, we call the model a convolutional autoencoder. Using a loss function we can quantify the difference between the input and the output, which we expect to be higher for anomalous observations. For both models, a loss threshold should be set for when to classify a sample as normal or abnormal [23].

A significant step towards deploying convolutional models e.g. CNNs and convolutional autoencoders on small devices such as smartphones, but also microcontrollers, are depthwise separable convolutions. These were first proposed in [29], and popularized in MobileNets [12].

4 Model Optimisations

Most disadvantages of tinyML that we listed in Section 1 come from microcontrollers being much less powerful than desktop or server computers. Therefore it is natural to apply optimisations to a tinyML pipeline that we want to run in microcontrollers. Section 3 explained that the focus of tinyML at this point seems to be NN models, so we will focus on optimisations for these models. Overall we describe six ways to optimise the performance of NN based models for tinyML. These are quantisation, pruning, clustering, knowledge distillation, removing operations and cascading architectures.

Quantisation. Most NNs represent their weights, biases and activations as 32-bit floats. This poses two problems for deploying them on microcontrollers. Firstly, not all microcontrollers have hardware support for floating-point units. Secondly, the many 32-bit values can take up a large part of the memory of microcontrollers. For example the Arduino Nano 33 BLE Sense, mentioned in Section 3, has 256 KB of RAM. That leaves room for 64.000 weights, biases, and activations. While that might sound like a lot, many modern NNs have much more. E.g. AlexNet and Resnet-50 both contain more than one million weights, biases and activations [3]. That is without even considering the memory required for the model structure, the input data, or the remaining application.

Fortunately, research has shown that it is possible to quantise NNs, while still retaining a good model [21]. For most hardware, operations using 8-bit integers are some of the fastest operations, and as such many tinyML models are quantised from 32-bit floats to 8-bit integers. This quantisation is typically done by taking the minimum and maximum 32-bit floating-point weights and mapping them and the intervals between them to 8-bit integers. Note that after full 8-bit integer quantisation, multiplying or adding two 8-bit integers can easily create an overflow situation. This is due to the minimum and maximum 32-bit floating-point weights being mapped to the minimum and maximum values for 8-bit integers. Consider that applying just the smallest multiplication or addition to the largest 32-bit floating-point weight after quantisation will result in an overflow. Therefore some approaches only quantise weights and biases (or only weights, as they grossly outnumber biases), and let the remaining activations (and biases) stay as 32-bit floats. A way to achieve full 8-bit integer quantisation is to compute the 8-bit integer computations and store the result in 16-bit/32-bit integers. This can then be scaled down to 8-bit integers again for the next computation. By using quantisation we can therefore reduce both the model size, increase inference speed, and make the model run on an even larger range of devices. The downside is a potential loss in accuracy [28]. This potential accuracy loss can be reduced using quantisation aware training. In this method, a model is trained with the knowledge that it will be quantised later [13]. Note that a 4 times reduction in the number of weights, biases and activations will make neither AlexNet nor Resnet-50 fit in the Arduino that we are considering. To achieve that we require further optimisations or smaller models.

Researchers have also been looking into further quantisation and even binarisation of NNs, which can further decrease the size by up to 32 times and inference time of the networks by up to 52 times [24].

Pruning. It is common in NNs that some weights are more relevant than others. After quantisation, some weights might even be zero and not contribute to the inference at all. In such cases, we can prune the connections associated with these weights. All incoming connections to a neuron might be pruned using such an approach. In that case, we can also prune that neuron and any outgoing connections. This will further reduce the model size and make it faster to compute. It is also an option to prune non-zero weight connections. In this case, some

rules should be set for when to prune a connection. This could e.g. be pruning connections when their associated weights are below a threshold [18]. Such an approach is used in [10], where pruning reduces the size of a NN by 9 to 13 times.

Clustering. An approach that is closely related to both quantisation and pruning is clustering. In this optimisation technique, weights are clustered into groups, where all weights in one group are assigned the same weight. Similarly to pruning, this technique reduces the model size, however, the computation is not sped up. The paper that initially introduced clustering claims that their approach reduced the size of a NN by 27 to 31 times [10]. This is after pruning has reduced the size by 9 to 13 times as reported above in Section 4.

Knowledge Distillation. Larger NN models are often better at learning the structure of a complicated dataset. However, as discussed earlier, it might be infeasible to deploy large models on microcontrollers. A solution is to "distil" the knowledge of a large model into a smaller model. This is known as knowledge distillation. The idea is to train a small model, not just using the ground truths but also using the predictions of a larger model [11]. Consider the following example. We want to create a small model that can classify the contents of an image. Normally this would be done by training the network to make the same classification as the ground truth labels. In knowledge distillation, we first train a larger model on our data. We then have the larger model give its classifications for all images in the dataset. Then we train the small classifier, not just to classify the ground truth, but to also make similar classifications as the larger model. This can be done by altering the loss function of the smaller model. Often the larger model is referred to as the "teacher", and the small model as the "student".

Removing Operations. A technique that is specific to the TFLM interpreter is the option to decide which NN operations the interpreter can execute. By removing operations from the interpreter, it is possible to reduce its size [4].

Cascading Architectures. Another approach to reducing the size and inference speed of a model is to split the model into two or more models of increasing size. Typically the idea is to use a small model as a filter before activating the larger model. E.g. when using a Google Home device, a small model is running locally, which listens for the "Hey Google" keywords. Once it detects these keywords, it sends the remaining speech to a larger model in the cloud to further process the request [32]. While the idea of cascading architectures is usually restricted to model size and alternative systems, research has looked into a cascading use of internal hardware in a system [33].

5 Input Optimisations

While there has been much research into optimising ML models for tinyML, there has, to our knowledge, been little work into optimising model inputs. Therefore

in this section, we describe preliminary work that we have done on optimising the input for a PdM tinyML model.

We choose to work with the ToyADMOS dataset, as it is the benchmarking dataset for tinyML. From the dataset, we use the recordings of one microphone from one case of the ToyConveyor part of the ToyADMOS dataset. This dataset is targeted towards unsupervised anomaly detection and contains normal and anomalous sound recordings. Due to this, and that tinyML is focused on NNs, we create a convolutional autoencoder model. The reason for choosing a convolutional autoencoder, above a standard autoencoder, is that it is common to generate images (spectrograms) from audio, and input this to the NN model [19]. We implement this model using TF and TFLM. We train the convolutional autoencoder on Mel spectrograms of the normal sound samples of the dataset. Thus the model learns to reconstruct a Mel spectrogram of a normal sound sample. After training the model, we mix a few normal Mel spectrograms (not used in training) and anomalous Mel spectrograms to create a test set. Generating a TFLM model from the convolutional autoencoder shows that the model takes up ∼172 KB before applying model Optimisations. This suggests that the model will be able to fit in the Arduino Nano 33 BLE Sense microcontroller.

The autoencoder architecture can be divided into an encoder and a decoder. The first layer of the encoder is a convolutional layer of 32 3x3 filters. The second layer is a max-pooling layer with a pooling size of 2x2. The third and fourth layers are a convolutional layer of 64 3x3 filters and a max-pooling layer, with a pooling size of 2x2, respectively. The first layer of the decoder is a convolutional layer with 32 3x3 filters. The second layer is an upsampling layer with a size of 2x2. The third and fourth layers of the decoder contain a convolutional layer of 16 3x3 filters and an upsampling layer of size 2x2. Lastly, the decoder contains a convolutional layer with one 3x3 filter to reproduce the spectrogram shape. All convolutional layers use the rectified linear unit activation function.

We evaluate the model using a Receiver Operating Characteristic (ROC) Area Under Curve (AUC) score calculated on the test set. A ROC AUC score of 1 means that there is a threshold for which the model can completely separate the normal and anomalous samples. A model that would take random actions, also known as a no-skill model, would on average get a ROC AUC score of 0.5.

At a sample rate of 24000 Hz of the sound files, we can train and run a model that in five out of five runs have a ROCAUC score of 1. We treat this sample rate as a benchmark and investigate the effects of reducing it. The idea is that the lower the sample rate, the lower the processing and storage capability is required to run inference on the files, which would be a useful optimisation for tinyML. We conduct experiments where we reduce the sample rate by a factor of two until reaching 375 Hz. Note that we do not cross-optimise the parameters of the Short Time Fourier Transform (STFT) used to create Mel spectrograms to the sampling frequency. The results are shown in Table 1, and are plotted in Figure 2 and 3 As this experiment is preliminary work, we measure the inference time on an M1 Pro chip with 10 CPU cores and 16 GPU cores. We expect to see a similar decrease in inference time on the Arduino Nano 33 BLE Sense.

Table 1. Mean and standard deviation of results from 5 training rounds.

Sample Rate (Hz)	ROC AUC score	Inference time (s)
24000	1.000 ± 0.0000	5.20 ± 0.022
12000	0.999 ± 0.0004	1.65 ± 0.015
6000	0.993 ± 0.0054	0.97 ± 0.041
3000	0.972 ± 0.0133	0.55 ± 0.011
1500	0.766 ± 0.0488	0.39 ± 0.013
750	0.649 ± 0.1110	0.28 ± 0.030
375	0.483 ± 0.0294	0.21 ± 0.001

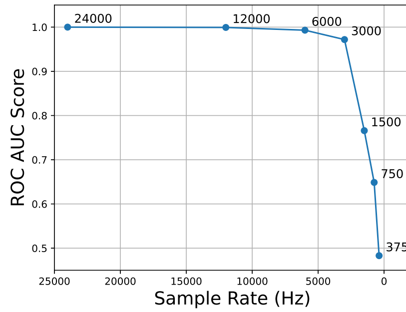

Fig. 2. AUC score by sample rate

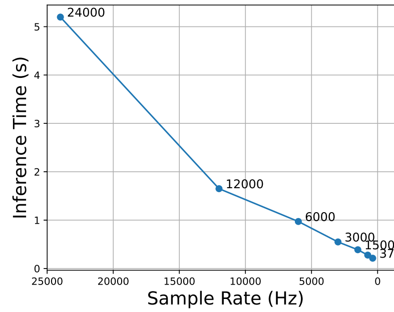

Fig. 3. Inference time by sample rate

The results suggest that we can reduce the sample rate from 24000 Hz to about 6000 Hz while still retaining a respectable ROC AUC score. This shows that we can reduce the input size by four times and speed up inference about five times by optimising the input in this example. The reduced inference time is a clear proxy for reduced computational requirements. By lowering the sample rate, we also reduce the dimensions of Mel spectrograms, and in turn reduce the input and output dimensions of the convolutional autoencoder. Thus both the input and model memory requirement is reduced when lowering the sample rate. We furthermore expect that these reductions translate to a reduction in energy consumption, both due to lower compute and memory requirements, but also due to lower sensing requirements at lower sampling rates.

6 Conclusion

In this paper, we presented the background behind deploying tinyML based PdM. We furthermore investigated the optimisations that can be used to achieve PdM using tinyML. We started by describing the model optimisations that are proposed in tinyML research. Lastly, we expressed our idea that optimisations should be considered throughout the ML pipeline, and that especially optimising

input to tinyML is a promising research direction. An example of input Optimisation was done for an industry-standard dataset for a convolutional autoencoder that fits in the memory of a tinyML device. The results suggest that there is a great potential for input optimisation to help achieve PdM using tinyML.

Acknowledgement. This work is supported by the Innovation Fund Denmark for the project DIREC (9142-00001B).

Resources. The source code used for the experiments is publicly accessible on GitHub: https://github.com/Ekhao/ToyADMOSTinyAutoencoder

References

1. ARM: Processor ip for the widest range of devices, https://www.arm.com/products/silicon-ip-cpu
2. Banbury, C., Reddi, V.J., Torelli, P., Jeffries, N., Kiraly, C., et al.: MLPerf Tiny Benchmark. In: Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1) (2021)
3. Bernstein, L., Sludds, A., Hamerly, R., Sze, V., Emer, J., Englund, D.: Freely scalable and reconfigurable optical hardware for deep learning. Scientific reports **11**(1), 1–12 (2021)
4. David, R., Duke, J., Jain, A., Janapa Reddi, V., Jeffries, N., Li, J., Kreeger, N., Nappier, I., Natraj, M., et al.: Tensorflow lite micro: Embedded machine learning for tinyml systems. Proc. Machine Learning and Systems **3**, 800–811 (2021)
5. Dutta, D.L., Bharali, S.: TinyML Meets IoT: A Comprehensive Survey. Internet of Things (Netherlands) **16** (12 2021). https://doi.org/10.1016/j.iot.2021.100461
6. edX: Professional certificate in tiny machine learning (tinyml), https://www.edx.org/professional-certificate/harvardx-tiny-machine-learning
7. Fafoutis, X., Marchegiani, L., Elsts, A., Pope, J., Piechocki, R., Craddock, I.: Extending the battery lifetime of wearable sensors with embedded machine learning. In: IEEE 4th World Forum on Internet of Things (WF-IoT). pp. 269–274 (2018)
8. tinyML Foundation: About us tinyml, https://www.tinyml.org/
9. Garofalo, A., Rusci, M., Conti, F., Rossi, D., Benini, L.: Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors. Philosophical Transactions of the Royal Society A **378**(2164), 20190155 (2020)
10. Han, S., Mao, H., Dally, W.J.: Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149 (2015)
11. Hinton, G., Vinyals, O., Dean, J., et al.: Distilling the knowledge in a neural network. arXiv preprint arXiv:1503.02531 **2**(7) (2015)
12. Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017)
13. Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., Kalenichenko, D.: Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 2704–2713 (2018)

14. Khan, A., Hammerla, N., Mellor, S., Plötz, T.: Optimising sampling rates for accelerometer-based human activity recognition. Pattern Recognition Letters **73**, 33–40 (2016)
15. Koizumi, Y., Saito, S., Uematsu, H., Harada, N., Imoto, K.: Toyadmos: A dataset of miniature-machine operating sounds for anomalous sound detection. In: IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA). pp. 313–317 (2019)
16. Kumar, A., Goyal, S., Varma, M.: Resource-efficient machine learning in 2 kb ram for the internet of things. In: Int. Conf. Machine Learning. pp. 1935–1944 (2017)
17. Lai, L., Suda, N., Chandra, V.: Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. arXiv preprint arXiv:1801.06601 (2018)
18. LeCun, Y., Denker, J., Solla, S.: Optimal brain damage. Advances in neural information processing systems **2** (1989)
19. Marchegiani, L., Newman, P.: Listening for sirens: Locating and classifying acoustic alarms in city scenes. IEEE Transactions on Intelligent Transportation Systems pp. 1–10 (2022). https://doi.org/10.1109/TITS.2022.3158076
20. Moons, B., Bankman, D., Yang, L., Murmann, B., Verhelst, M.: Binareye: An always-on energy-accuracy-scalable binary cnn processor with all memory on chip in 28nm cmos. In: IEEE Custom Integrated Circuits Conf (CICC). pp. 1–4 (2018)
21. Nagel, M., Fournarakis, M., Amjad, R.A., Bondarenko, Y., van Baalen, M., Blankevoort, T.: A white paper on neural network quantization. arXiv preprint arXiv:2106.08295 (2021)
22. Purohit, H., Tanabe, R., Ichige, K., Endo, T., Nikaido, Y., Suefusa, K., Kawaguchi, Y.: Mimii dataset: Sound dataset for malfunctioning industrial machine investigation and inspection. arXiv preprint arXiv:1909.09347 (2019)
23. Ran, Y., Zhou, X., Lin, P., Wen, Y., Deng, R.: A survey of predictive maintenance: Systems, purposes and approaches. arXiv preprint arXiv:1912.07383 (2019)
24. Rastegari, M., Ordonez, V., Redmon, J., Farhadi, A.: Enabling ai at the edge with xnor-networks. Communications of the ACM **63**(12), 83–90 (2020)
25. Ray, P.P.: A review on tinyml: State-of-the-art and prospects. Journal of King Saud University - Computer and Information Sciences (2021)
26. Sanchez-Iborra, R., Skarmeta, A.F.: Tinyml-enabled frugal smart objects: Challenges and opportunities. IEEE Circuits and Systems Magazine **20**, 4–18 (7 2020)
27. Saxena, A., Goebel, K.: Turbofan engine degradation simulation data set. NASA Ames Prognostics Data Repository pp. 1551–3203 (2008)
28. Siang, Y.Y., Ahamd, M.R., Abidin, M.S.Z.: Anomaly detection based on tiny machine learning: A review. Open International Journal of Informatics **9**(Special Issue 2), 67–78 (2021)
29. Sifre, L., Mallat, S.: Rigid-motion scattering for texture classification. arXiv preprint arXiv:1403.1687 (2014)
30. TensorFlow: Tensorflow lite for microcontrollers, https://www.tensorflow.org/lite/microcontrollers
31. Tzimpragos, G., Madhavan, A., Vasudevan, D., Strukov, D., Sherwood, T.: In-sensor classification with boosted race trees. Communications of the ACM **64**(6), 99–105 (2021)
32. Warden, P., Situnayake, D.: TinyML. O'Reilly Media, Incorporated (2019)
33. Zalewski, P., Marchegiani, L., Elsts, A., Piechocki, R., Craddock, I., Fafoutis, X.: From bits of data to bits of knowledge—an on-board classification framework for wearable sensing systems. Sensors **20**(6), 1655 (2020)