Sustainable Deep Learning for Fault Diagnosis: A Comparative Study of Different Compression Techniques in Embedded Platforms

Matheus Oliveira
SENAI CIMATEC University Center
Salvador, Brazil
email: matheus.guimaraes@fbter.org.br

Rafael Macieira
SENAI ISI TICs
Recife, Brazil
email: rafael.macieira@sistemafiepe.org.br

Alex Santos
SENAI CIMATEC University Center
Salvador, Brazil
email: alex.santos@fieb.org.br

Regis Cardoso
SENAI ISI SE
Florianópolis, Brazil
UNISINOS University
São Leopoldo, Brazil
email: regis.cardoso@sc.senai.br

Arthur Paim
SENAI CIMATEC University Center
Salvador, Brazil
email:arthur.paim@fieb.org.br

Lilian Guarieiro
SENAI CIMATEC University Center
Salvador, Brazil
email:lilian.guarieiro@fieb.org.br

Lucas Cambuim
SENAI ISI TICs
Recife, Brazil
email: lucas.cambuim@sistemafiepe.org.br

Helaine Neves
SENAI CIMATEC University Center
Salvador, Brazil
email: helaine.neves@fieb.org.br

Erick G. Sperandio Nascimento
Surrey Institute for People-Centred AI, University of Surrey
Guildford GU2 7XH, UK
email: erick.sperandio@surrey.ac.uk
SENAI CIMATEC University Center
Salvador, Brazil

***Abstract*—This paper aims to develop a comparative analysis of different compression techniques applied to a state-of-the-art deep learning (DL) model for machinery fault diagnosis on different embedded platforms, aiming at fostering more sustainable DL practices towards Industry 5.0 standards, making these solutions more energy efficient and thus more environmentally friendly. The evaluation uses the Machinery Fault Database (MaFaulDa) and the Predictive Maintenance Model with a Convolutional Neural Network (PdM-CNN). We measured the model's average inference time, size, and evaluation metrics for each type of technique applied for compression. The compression technique that led to the shortest inference time was Knowledge Distillation. The research is relevant for the TinyML (Tiny Machine Learning) field, as it presents the analysis of a new model in the context of TinyML, showing how it performs on various platforms under different model compression techniques.**

***Index Terms*—TinyML, Benchmarking, Deep Learning, Fault Diagnosis, Sustainable**

I. Introduction

Deep learning (DL) models perform many complicated operations, which makes them consume a lot of energy. Regarding their environmental impact, the situation is concerning [1]. A field of study that is central in the reduction of the negative effects of DL models is tiny machine learning. TinyML can be defined as the set of machine learning-aware architectures, frameworks, techniques, tools, and approaches that enable the performance of a variety of tasks in the mW (Miliwatt) power range setting (or below) while being suitable for implementation in large-scale use cases [2]. Through the use of TinyML, it is possible to compress machine learning models to embed them in microcontrollers. The use of an embedded machine learning model has been discussed since 2006 [3]. With the definition of the concept of industry 4.0, other work fronts began to be discussed, including the use of the Wisdom Web of Things (W2T). Through W2T, the increased use of IoT systems and the possibility of increasing the use of these devices over the years [4] were discussed. Consequently, devices would have to evolve and use artificial intelligence embedded within themselves.

The use of embedded artificial intelligence is the future of IoT [5]. This will result in a reduction in the load for executing algorithms in the cloud or on local servers. Through the use of embedded AI, it is possible to process data and infer results directly on devices locally, without the need for any type of connection [6].

The use of embedded AI brings a series of advantages to the most diverse applications. Such as real-time decision-making, as the algorithms are integrated directly into the devices. Greater data privacy and security, as information is processed locally without the need to transfer to other types of servers, eliminating the need for constant communication with external systems [7]. Additionally, embedded AI enables resource optimization, as algorithms can be adapted to run efficiently on specific hardware, making the most of available resources such as low-power processors and specialized graphics processing units [8].

However, the use of embedded machine learning models brings many challenges. One of these challenges is the limitation of computing resources, especially on embedded devices with less powerful hardware. This can result in restrictions on the complexity and processing power of AI algorithms and affect their effectiveness and accuracy. Many of these challenges are open in the area; two other important ones are hardware heterogeneity and the need for new machine learning models

[2]. Based on such problems, a question arose: How does a Convolutional Neural Network for Predictive Maintenance (PdM-CNN) [9] work on different devices and under different compression techniques? To answer this question and face the challenges mentioned above, this research was carried out.

The section 2 presents the methods, development platforms used for the tests, dataset description, pipeline of test and quatized methods. The results are presented in section 3 and finally the conclusion in section 4.

II. Methods

In the development of this research, we compared results from the application of a Convolutional Neural Network for Predictive Maintenance (PdM-CNN) on the following devices: a desktop with an Intel(R) Core(TM) i5-10400F processor, a laptop with an Intel(R) Core(TM) i5-11300H processor, a High Performance Computing cluster node with an Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz Processor, a STM32MP157F-DK2, a Raspberry Pi 4 model B and a Jetson Nano Developer Kit. For this, we tested different techniques for model size reduction: Dynamic Range Quantization with 8-bit precision, Integer Only Full Integer Quantization with 8-bit precision, Float16 Quantization and Knowledge Distillation. Device specifications are presented in Table I.

TABLE I
Devices used for this research.

Device	Processor	Architecture	Frequency
Desktop	Intel(R) Core(TM) i5-10400F	x86-64	2.90 GHz
Laptop	Intel(R) Core(TM) i5-11300H	x86-64	2.60 GHz
HPC-CPU	Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz	x86-64	2.40GHz
STM32	Dual Arm® Cortex®-A7	ARMv7-A	800 MHz
Raspberry	Quad-core ARM® Cortex®-A72	ARMv8-A	1.5GHz
Jetson nano	Quad-core ARM® Cortex®-A57	ARMv8-A	1.43 GHz

Due to the need to compress models to accommodate them on low-power devices, it is common for models to lose performance. We sought to evaluate this performance and verify how significant this loss is for predictive maintenance applications.

Evaluating the performance of embedded machine learning models on different development platforms is very important to determine the viability and effectiveness of these solutions in real-world applications. Using development platforms to test embedded models is critical to ensuring that these systems meet the performance, accuracy, and reliability requirements in their specific applications. The use of different development platforms means we have a great diversity of processors, covering the vast majority of microcontrollers available on the market.

To perform the study, a modified version of the PdM-CNN model [9] underwent 4 different types of model reduction. After this, all models were converted to the tflite format using the TensorFlow Lite Converter. The tflite models were benchmarked based on the Machinery Fault Database (MaFaulDa) [10], [11], developed by the Federal University of Rio de Janeiro (UFRJ), which was also used to train the PdM-CNN model. With the different versions of the model, there were benchmarks to measure the average inference time and also get the evaluation metrics for each version so they could be compared. The code for these benchmarks was developed in the Python language for all devices.

A. Dataset Description

The MaFaulDa dataset is a common reference in predictive maintenance projects [9], [12], [13]. This database contains 1951 multivariate time series collected by sensors on a SpectraQuest Machinery Fault Simulator (MFS) Alignment, Balance, and Vibration (ABVT). 1951 has six possible simulated states: normal function, imbalance fault, horizontal and vertical misalignment faults, and inner and outer bearing faults [10], [11]. The experimental setup used to collect data from the MaFaulDa dataset is presented in Figure 1.

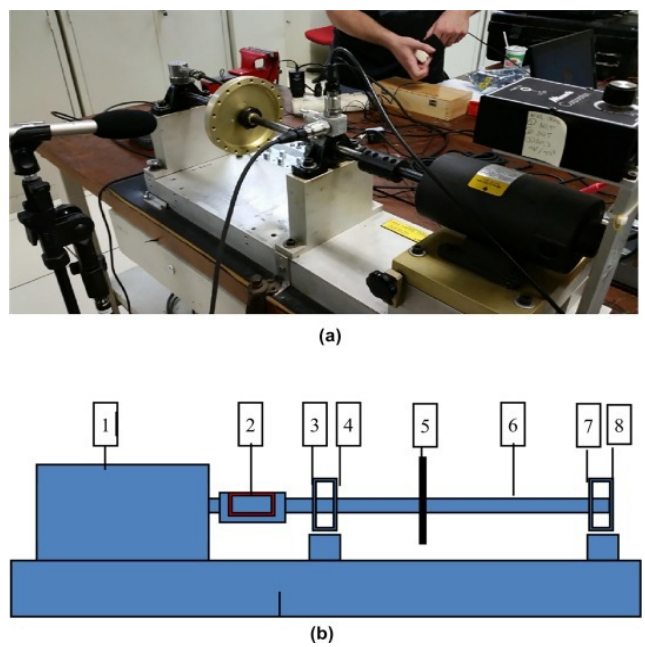

Fig. 1. MaFaulDa experimental setup.

Data were collected using two sets of accelerometers (102 mV/g sensitivity) positioned longitudinally, horizontally, and vertically, as well as a tachometer for rotor speed and a microphone for noise measurement. The vibration signals were captured at 51.2 kHz using the National Instruments NI 9234 data collection system.

1) Normal function: In this class, data were acquired without introducing any errors. The results were collected from 49 test situations with rotation speeds ranging from 737 to 3686 rpm. During the experiments, the speed was fixed at roughly 60 rpm increments.

2) Imbalance: This class is composed of data from tests conducted with various loads installed in the rotor, such as 6, 10, 15, 20, 25, 30, and 35 grams. The database contained 333 different scenarios. The testing platform vibrated excessively, so

no data were obtained when a load equal to or greater than 30 g was applied and the speed was more than 3300 rpm.

3) Horizontal misalignment: To form this class, data of a horizontal motor shaft tests at 0.5, 1.0, 1.5, and 2.0 mm intervals were obtained. The tests used the same rotations as in normal operation. It consists of 197 scenarios.

4) Vertical misalignment: The data gathered during vertical motor shaft tests (0.51, 0.63, 1.27, 1.4, 1.78, and 1.9 mm) formed this class, which featured 301 scenarios. The tests used the same rotations for each vertical step as they would during normal operation.

5) Bearing faults: Data for three types of bearing faults make up this one. There were outer races, ball elements, and cage defects. The experiments involved testing one defective bearing at two support positions: the drive end (position 3 in Figure 1a) and the fan end (position 7 in Figure 1b). The mass of the system dampened the effect of bearing defects, making them almost imperceptible in the absence of imbalance. To achieve a noticeable effect, three masses weighing 6, 10, and 20 g were added. All tests were conducted using the same rotations as during normal operation. The database consists of 558 underhang and 513 overhang scenarios.

B. Evaluation Pipeline

The evaluation pipeline implemented in this study is illustrated in Figure 2.

The MaFaulDa dataset is composed of several sensors in different positions. The proposed model uses only the signals captured by three unidirectional accelerometers, similar to a triaxial accelerometer installed at the drive end, as this is the most commonly used monitoring configuration in industrial processes. The sensor is located in position 3.

To improve the model performance, the Fast Fourier Transform (FFT) was extracted from each vibration sensor data and used as the input vector for training, validation, and testing. Resource extraction significantly reduces the input vector size compared to the number of samples in a time series. This step improves the discrimination of failure classes compared to the original time series.

Soon after, the data were normalized using Sklearn's (scikit-learn) MinMax Scaler method. 8580 samples were extracted from the sequences of FFT through slicing in smaller time windows. 5550 were used for training, 1220 for validation, and 1810 for testing.

The Predictive Maintenance model with Convolutional Neural Network (PdM-CNN) framework was created to automatically identify faults in rotating machinery. The model utilized ReLU (Rectified Linear Unit) and weight normalization as its first step. The softmax and loss functions were used to optimize faults in rotating machinery.

The original PdM-CNN model was developed for classifying 30 classes; the version used in this article classifies only 5: normal function, imbalance, horizontal parallel misalignment, bearing cage fault on the underhang position, and bearing outer race fault on the underhang position. Its architecture can be seen in Table II.

Data input for testing was done as follows: Python code was used to loop over 1810 samples extracted from the sequences of the MaFaulDa Dataset, having gone through the Fast Fourier Transform and data normalization. In each case, inference was performed using the model in its baseline format and under different compression techniques.

TABLE II
NEURAL NETWORK ARCHITECTURE FOR THE ORIGINAL PDM-CNN.
TOTAL PARAMS: 303,907. TRAINABLE PARAMS: 303,905. NON-TRAINABLE PARAMS: 0. OPTIMIZER PARAMS: 2

Layer (type)	Output Shape	Param #
conv1d_4 (Conv1D)	(None, 2491, 100)	3100
conv1d_5 (Conv1D)	(None, 2482, 100)	100100
max_pooling1d_1 (MaxPooling1D)	(None, 827, 100)	0
conv1d_6 (Conv1D)	(None, 818, 100)	100100
conv1d_6 (Conv1D)	(None, 809, 100)	100100
global_average_pooling1d_1 (GlobalAveragePooling1D)	(None, 100)	0
dropout_1 (Dropout)	(None, 100)	0
dense_1 (Dense)	(None, 5)	505

TABLE III
NEURAL NETWORK ARCHITECTURE FOR THE KNOWLEDGE DISTILLATION (KD) MODEL.
TOTAL PARAMS: 3,395. TRAINABLE PARAMS: 3,395. NON-TRAINABLE PARAMS: 0.

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 2491, 10)	310
conv1d_1 (Conv1D)	(None, 2482, 10)	1010
max_pooling1d_1 (MaxPooling1D)	(None, 827, 10)	0
conv1d_2 (Conv1D)	(None, 818, 10)	1010
conv1d_3 (Conv1D)	(None, 809, 10)	1010
global_average_pooling1d (GlobalAveragePooling1D)	(None, 10)	0
dropout (Dropout)	(None, 10)	0
dense (Dense)	(None, 5)	55

Finally, the performance of the models was evaluated using the F1-Score, the ROC AUC (area under the receiver operating characteristic curve) metric, PRC AUC (Precision-Recall area under the curve) metric, the Precision metric,the Recall metric, the Specificity metric, and the Accuracy metric. The inference time was also measured.

C. TensorFlow Lite

TensorFlow Lite is a collection of tools that enables on-device machine learning by helping developers run their models on mobile, embedded, and edge devices [14]. The representation of a trained model using the tools of TensorFlow Lite is made using a format known as FlatBuffers (identified by the .tflite file extension). Which provides a series of benefits over TensorFlow's model format, such as reduced size and faster inference, enabling TensorFlow Lite to execute efficiently on devices with limited compute and memory resources [14].

D. Model Compression Techniques

1) Integer Only Full Integer Quantization (QI8): In this method, for full integer quantization, you need to calibrate all floating-point tensors in the model. Variable tensors such as model input, activation functions (outputs of intermediate layers), and model output cannot be calibrated unless we run a few inference cycles. As a result, the converter requires a representative dataset to calibrate them. This dataset can be a small subset of the training or validation data. This technique ensures compatibility with integer-only devices, forcing full integer quantization for all operations, including the input and

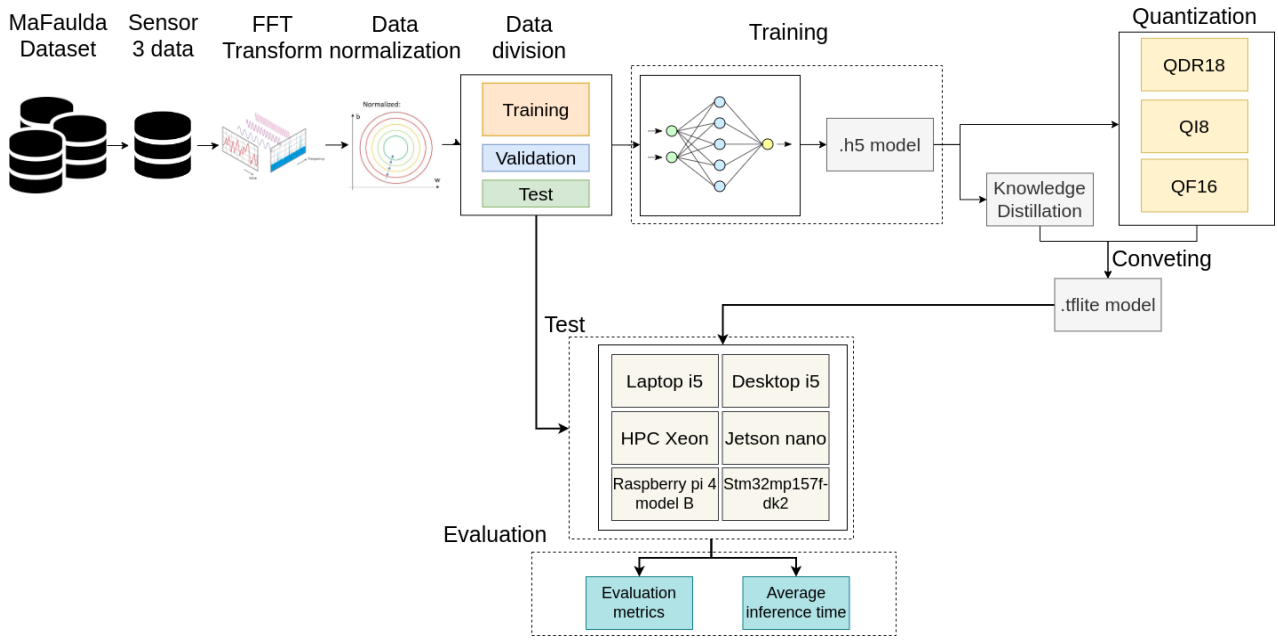

Fig. 2. Proposed evaluation pipeline. Complete pipeline for the development of this research.

output. It converts 32-bit floating-point numbers to the nearest 8-bit fixed-point numbers [14], [15].

2) Dynamic Range Quantization (QDRI8): In this technique, the weights are statically quantized post-training from floating point to integer,which provides 8-bit precision. The activation functions are quantized dynamically at inference time. Dynamic-range operators dynamically quantize activation functions based on their range to 8-bits and perform computations with 8-bit weights and activation functions. The model weights are not retrained to compensate for quantization-induced errors. The activation functions are always stored in floating points, so the increased speed of dynamic-range ops is less than a full fixed-point computation. For operations that support quantized kernels, the activation functions are quantized to 8 bits of precision dynamically prior to processing and are dequantized to float precision after processing. It provides reduced memory usage and faster computation without you having to provide a representative dataset for calibration [14].

3) Float16 Quantization (QF16): This approach reduces the size of a floating point model by quantizing the weights to float16, the IEEE standard for 16-bit floating point numbers. All weights become half of their original size. By default, a float16 quantized model will dequantize the weight values to float32 when run on the CPU [14].

4) Knowledge Distillation(KD): A model compression technique, in which a small model (student) is trained to match a large pre-trained model (teacher). Knowledge is transferred from the teacher model to the student by minimizing a loss function, aimed at matching softened teacher logits as well as ground-truth labels. The logits are softened by applying a ”temperature” scaling function in the softmax, revealing inter-class relationships learned by the teacher [16], [17]. The student’s model architecture is presented in Table III.

III. Results and Discussion

Analyzing the data in Table V of evaluation metrics, we observed that the metrics for the different methods (Baseline, QDRI8, QF16, QI8, KD) are very close, suggesting consistency in the results. While there are small variations between methods, as seen in the F1 and ROC AUC metrics, these variations can be insignificant depending on the project context. All evaluated metrics (F1, ROC AUC, PRC AUC, Precision, Recall, Specificity, and Accuracy) present high results, indicating robust performance across all methods. High sensitivity and specificity stand out, indicating effectiveness in identifying positive and negative samples, respectively. Precision and accuracy are also consistently high, suggesting reliable and consistent results.

When analyzing the Figure 3 of the model submitted to different techniques, we observed a significant reduction in the size of the models after applying the compression techniques. All reduced models, have considerably smaller sizes compared to the baseline model, which is 1,222.696 Kilobytes. The reduction in size varies significantly between reduction techniques. For example, the QDRI8 model is 318.616 Kilobytes, while the QF16 model is 616.348 Kilobytes, showing that the QF16 technique results in a larger model size compared to QDRI8. The Knowledge Distillation one was the smallest, having only 20.116 Kilobytes of size.

By analyzing Table IV of average inference times for different compression techniques on various devices, we observed that each compression technique has a distinct impact on the time required to perform inferences. We note that, in general, compression techniques reduce inference time relative to the baseline model. However, the magnitude of this reduction varies depending on the compression technique and device used.

For example, for the desktop device, the QDRI8, QF16, and QI8 techniques show slightly higher inference times compared to the baseline model. This suggests that although these techniques can reduce model size, they can increase the time

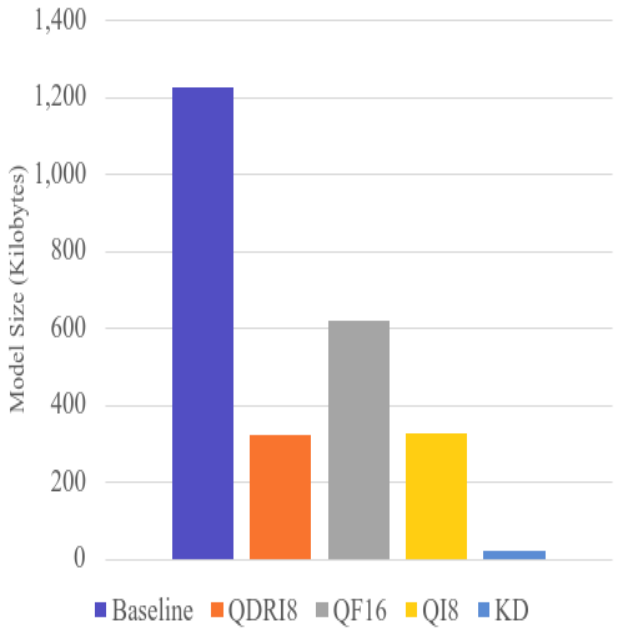

Fig. 3. Histogram plot presenting the size of each compression technique compared to the baseline version of the model.

TABLE IV
AVERAGE INFERENCE TIME (MS) FOR EACH DEVICE UNDER THE DIFFERENT COMPRESSION TECHNIQUES

Devices	Baseline	QDRI8	QF16	QI8	KD
Desktop	14.69	17.33 ↑ (18.00%)	15.13 ↑ (3.02%)	7.99 ↓ (45.61%)	0.39 ↓ (97.31%)
Laptop	7.77	10.98 ↑ (41.41%)	7.60 ↓ (2.11%)	5.81 ↓ (25.21%)	0.34 ↓ (95.60%)
HPC-CPU	7.34	12.44 ↑ (69.49%)	7.38 ↑ (0.56%)	7.47 ↑ (1.68%)	0.39 ↓ (94.67%)
STM32	784.95	499.38 ↓ (36.38%)	772.23 ↑ (1.62%)	393.08 ↓ (49.92%)	22.86 ↓ (97.09%)
Raspberry	81.77	57.22 ↓ (30.03%)	83.68 ↑ (2.33%)	54.97 ↓ (32.77%)	3.32 ↓ (95.94%)
Jetson nano	148.52	98.55 ↓ (33.64%)	150.65 ↑ (1.44%)	93.05 ↓ (37.35%)	5.13 ↓ (96.54%)

↑ Percentage Growth ↓ Percentage Reduction

required to perform inferences on this specific device. On the other hand, for the laptop device, different compression techniques result in lower inference times than the baseline model, indicating an improvement in inference performance.

IV. CONCLUSION

Analyzing the metrics evaluated, reliable performance can be seen and can be used safely in different contexts. The amount of model size reduction varies significantly between techniques, indicating that different methods produce different compression results, which may be interesting for devices with low storage memory. When choosing the compression technique, one must take into account not only the reduction in model size but also the impact on inference time, especially on different types of devices with different processing capabilities.

The ideal compression technique for the PdM-CNN model is Knowledge Distillation, as it provides the smallest model size with the smallest average inference time, while keeping metrics very close to the baseline, with very low reduction on the metrics. Thus, this research provides good evidence on how a convolutional neural network might perform on different MPU devices using different compression techniques. Although this study is focusing on only one model for predictive maintenance, its methodology can be applied to other similar cases. In future works, we intend to compare more models of different architectures using different databases and different compression techniques. Furthermore, we will seek to evaluate the impact of clock frequency in inference time and quality. A comparison of the energy consumption of each device relative to each compression technique will also be done.

V. ACKNOWLEDGMENTS

We thank the National Council of Scientific and Technological Development (CNPq) for partially fund this work through the CNPq Universal project 405706/2021-9. We also thank the Supercomputing Center for Industrial Innovation (CS2i) from SENAI CIMATEC, for providing HPC infrastructure support, and the Surrey Institute for People-Centred AI from the University of Surrey (UK) for their institutional support. Erick G. Sperandio Nascimento is a CNPq technological development fellow (Proc. 308963/2022-9).

TABLE V
EVALUATION METRICS FOR EACH DEVICE UNDER THE DIFFERENT COMPRESSION TECHNIQUES

Device	**Compression Technique**	**F1**	**ROC AUC**	**PRC AUC**	**Precision**	**Recall**	**Specificity**	**Accuracy**
Desktop	Baseline	0.9481	0.9647	0.9089	0.9590	0.9391	0.9903	0.9852
Desktop	QDRI8	0.9485 ↑ (0.042%)	0.9655 ↑ (0.083%)	0.9094 ↑ (0.055%)	0.9577 ↓ (0.135%)	0.9406 ↑ (0.159%)	0.9903	0.9852
Desktop	QF16	0.9481	0.9647	0.9089	0.9590	0.9391	0.9903	0.9852
Desktop	QI8	0.9495 ↑ (0.147%)	0.9653 ↑ (0.062%)	0.9115 ↑ (0.286%)	0.9612 ↑ (0.229%)	0.9401 ↑ (0.106%)	0.9906 ↑ (0.030%)	0.9856 ↑ (0.040%)
Desktop	KD	0.9291 ↓ (2.004%)	0.9601 ↓ (0.476%)	0.8790 ↓ (3.289%)	0.9273 ↓ (3.305%)	0.9352 ↓ (0.415%)	0.9850 ↓ (0.535%)	0.9757 ↓ (0.964%)
Laptop	Baseline	0.9481	0.9647	0.9089	0.9590	0.9391	0.9903	0.9852
Laptop	QDRI8	0.9485 ↑ (0.042%)	0.9655 ↑ (0.083%)	0.9094 ↑ (0.055%)	0.9577 ↓ (0.135%)	0.9406 ↑ (0.159%)	0.9903	0.9852
Laptop	QF16	0.9481	0.9647	0.9089	0.9590	0.9391	0.9903	0.9852
Laptop	QI8	0.9495 ↑ (0.147%)	0.9653 ↑ (0.062%)	0.9115 ↑ (0.286%)	0.9612 ↑ (0.229%)	0.9401 ↑ (0.106%)	0.9906 ↑ (0.030%)	0.9856 ↑ (0.040%)
Laptop	KD	0.9291 ↓ (2.004%)	0.9601 ↓ (0.476%)	0.8790 ↓ (3.289%)	0.9273 ↓ (3.305%)	0.9352 ↓ (0.415%)	0.9850 ↓ (0.535%)	0.9757 ↓ (0.964%)
HPC-CPU	Baseline	0.9481	0.9647	0.9089	0.9590	0.9391	0.9903	0.9852
HPC-CPU	QDRI8	0.9485 ↑ (0.042%)	0.9655 ↑ (0.083%)	0.9094 ↑ (0.055%)	0.9577 ↓ (0.135%)	0.9406 ↑ (0.159%)	0.9903	0.9852
HPC-CPU	QF16	0.9481	0.9647	0.9089	0.9590	0.9391	0.9903	0.9852
HPC-CPU	QI8	0.9495 ↑ (0.147%)	0.9653 ↑ (0.062%)	0.9115 ↑ (0.286%)	0.9612 ↑ (0.229%)	0.9401 ↑ (0.106%)	0.9906 ↑ (0.030%)	0.9856 ↑ (0.040%)
HPC-CPU	KD	0.9291 ↓ (2.004%)	0.9601 ↓ (0.476%)	0.8790 ↓ (3.289%)	0.9273 ↓ (3.305%)	0.9352 ↓ (0.415%)	0.9850 ↓ (0.535%)	0.9757 ↓ (0.964%)
STM32	Baseline	0.9481	0.9647	0.9089	0.9590	0.9391	0.9903	0.9852
STM32	QDRI8	0.9485 ↑ (0.042%)	0.9655 ↑ (0.083%)	0.9094 ↑ (0.055%)	0.9577 ↓ (0.135%)	0.9406 ↑ (0.159%)	0.9903	0.9852
STM32	QF16	0.9481	0.9647	0.9089	0.9590	0.9391	0.9903	0.9852
STM32	QI8	0.9495 ↑ (0.147%)	0.9653 ↑ (0.062%)	0.9115 ↑ (0.286%)	0.9612 ↑ (0.229%)	0.9401 ↑ (0.106%)	0.9906 ↑ (0.030%)	0.9856 ↑ (0.040%)
STM32	KD	0.9291 ↓ (2.004%)	0.9601 ↓ (0.476%)	0.8790 ↓ (3.289%)	0.9273 ↓ (3.305%)	0.9352 ↓ (0.415%)	0.9850 ↓ (0.535%)	0.9757 ↓ (0.964%)
Raspberry	Baseline	0.9481	0.9647	0.9089	0.9590	0.9391	0.9903	0.9852
Raspberry	QDRI8	0.9485 ↑ (0.042%)	0.9655 ↑ (0.083%)	0.9094 ↑ (0.055%)	0.9577 ↓ (0.135%)	0.9406 ↑ (0.159%)	0.9903	0.9852
Raspberry	QF16	0.9481	0.9647	0.9089	0.9590	0.9391	0.9903	0.9852
Raspberry	QI8	0.9495 ↑ (0.147%)	0.9653 ↑ (0.062%)	0.9115 ↑ (0.286%)	0.9612 ↑ (0.229%)	0.9401 ↑ (0.106%)	0.9906 ↑ (0.030%)	0.9856 ↑ (0.040%)
Raspberry	KD	0.9291 ↓ (2.004%)	0.9601 ↓ (0.476%)	0.8790 ↓ (3.289%)	0.9273 ↓ (3.305%)	0.9352 ↓ (0.415%)	0.9850 ↓ (0.535%)	0.9757 ↓ (0.964%)
Jetson nano	Baseline	0.9481	0.9647	0.9089	0.9590	0.9391	0.9903	0.9852
Jetson nano	QDRI8	0.9485 ↑ (0.042%)	0.9655 ↑ (0.083%)	0.9094 ↑ (0.055%)	0.9577 ↓ (0.135%)	0.9406 ↑ (0.159%)	0.9903	0.9852
Jetson nano	QF16	0.9481	0.9647	0.9089	0.9590	0.9391	0.9903	0.9852
Jetson nano	QI8	0.9495 ↑ (0.147%)	0.9653 ↑ (0.062%)	0.9115 ↑ (0.286%)	0.9612 ↑ (0.229%)	0.9401 ↑ (0.106%)	0.9906 ↑ (0.030%)	0.9856 ↑ (0.040%)
Jetson nano	KD	0.9291 ↓ (2.004%)	0.9601 ↓ (0.476%)	0.8790 ↓ (3.289%)	0.9273 ↓ (3.305%)	0.9352 ↓ (0.415%)	0.9850 ↓ (0.535%)	0.9757 ↓ (0.964%)

↑ Percentage Growth ↓ Percentage Reduction

REFERENCES

[1] Dhar, P. “The carbon impact of artificial intelligence.”, Nat Mach Intell 2, 423–425 (2020). https://doi.org/10.1038/s42256020-0219-9.

[2] P. P. Ray, “A review on TinyML: State-of-the-art and prospects,” Journal of King Saud University - Computer and Information Sciences, vol. 34, no. 4, pp. 1595–1623, Apr. 2022, doi: 10.1016/j.jksuci.2021.11.019.

[3] SCHOITSCH, E.; SKAVHAUG, A. “Embedded intelligence.” European Research Consortium for Informatics and Mathematics, v. 67, p. 14–15, 2006.

[4] GUO, B. et al. “From the internet of things to embedded intelligence.” World Wide Web, Springer, v. 16, p. 399–420, 2013.

[5] MANAVALAN, E.; JAYAKRISHNA, K.“A review of internet of things (iot) embedded sustainable supply chain for industry 4.0 requirements.” Computers e industrial engineering, Elsevier, v. 127, p. 925–953, 2019.

[6] LI, H.; OTA, K.; DONG, M. “Learning iot in edge: Deep learning for the internet of things with edge computing.” IEEE network, IEEE, v. 32, n. 1, p. 96–101, 2018.

[7] HOWARD, A. G. et al. “Mobilenets: Efficient convolutional neural networks for mobile vision applications.” arXiv preprint arXiv:1704.04861, 2017.

[8] SZE, V. et al. “Efficient processing of deep neural networks.” [S.l.]: Springer, 2020.

[9] Souza, Roberto M., et al. ”Deep learning for diagnosis and classification of faults in industrial rotating machinery.” Computers & Industrial Engineering 153 (2021): 107060.

[10] MAFAULDA:: Machinery Fault Database [Online]. Available:https://www02.smt.ufrj.br/~offshore/mfs/page01.html.

[11] Ribeiro, F. M. L. ”MaFaulDa-machinery fault database.” Signals, Multimedia, and Telecommunications Laboratory (2016).

[12] Wang, Youming, Zhao Xiao, and Gongqing Cao. ”A convolutional neural network method based on Adam optimizer with power-exponential learning rate for bearing fault diagnosis.” Journal of Vibroengineering 24.4 (2022): 666-678.

[13] MOSTAFAVI, Atabak, and Andreas FRIEDMANN. ”Application of Multi-Label Convolutional Neural Networks on Enhancement of Reliability of Classification, Case Study on Bearing Fault Diagnostics.”

[14] “TensorFlow,” TensorFlow. Accessed: May, 26, 2024, [Online]. Available:https://www.tensorflow.org/

[15] B. Jacob et al., “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference,” Jun. 2018, doi: 10.1109/cvpr.2018.00286.

[16] K. Team, “Keras documentation: Knowledge Distillation.” Accessed: Jun, 7, 2024, [Online]. Available:https://keras.io/examples/vision/knowledge_distillation/.

[17] HINTON, Geoffrey; VINYALS, Oriol; DEAN, Jeff. “Distilling the knowledge in a neural network.” arXiv preprint arXiv:1503.02531, 2015.