DTU Compute
Department of Applied Mathematics and Computer Science

Thesis title

by Mohammad Amin Hasanpour

PhD Thesis

Thesis title
by Mohammad Amin Hasanpour

PhD Thesis
January, 2026

By
Mohammad Amin Hasanpour

Cover photo: Vibeke Hempler, 2012
Published by: DTU, Department of Applied Mathematics and Computer Science, Richard Petersens Plads, Building 322, 2800 Kgs. Lyngby Denmark
www.compute.dtu.dk
ISSN: [0000-0000] (electronic version)
ISBN: [000-00-0000-000-0] (electronic version)

ISSN: [0000-0000] (printed version)
ISBN: [000-00-0000-000-0] (printed version)

English Abstract

English abstract goes here.

Dansk Abstrakt

Danish abstract goes here.

Acknowledgements

Acknowledgements go here.

Tool Acknowledgements

In writing this thesis, I have been making use of Generative AI services such as Grammarly and ChatGPT to improve writing and make grammatical corrections.

Contents

1 Introduction

Introduction goes here.

2 Paper A: Pump Cavitation Detection with Machine Learning: A Comparative Study of SVM and Deep Learning

2.1 Paper Abstract

In the pursuit of enhancing industrial pump reliability and efficiency, this paper addresses the challenging issue of pump cavitation detection through the innovative application of Machine Learning (ML) techniques. Cavitation, a prevalent problem in pumps, significantly compromises their performance, causing damage and operational inefficiencies. Traditionally, cavitation detection has relied on numerical analysis and signal processing methods, which, despite their merit, often fall short in real-world applications due to their requirement for extensive domain knowledge and controlled operational conditions. This study diverges from conventional approaches by harnessing the power of ML to predict cavitation occurrences in pumps under varying real-world conditions with high accuracy.

We present an analysis of a cavitation dataset compiled by the Danish pump manufacturer Grundfos, which includes vibration data from 297 experiments on seven different pumps, using both traditional ML models, specifically Support Vector Machine (SVM), and advanced Deep Learning (DL) techniques. Our methodology includes a detailed examination of the dataset, feature engineering, target definition, problem formulation, model design, and rigorous model testing on target hardware. Remarkably, our study not only demonstrates that ML models, particularly DL models, can adaptively and accurately predict cavitation but also emphasizes the importance of testing these models on target hardware to ensure their practical applicability. This work is accompanied by an open-source implementation.

2.2 Introduction

Pumps are indispensable in various industrial processes, including water treatment, oil and gas production, and manufacturing. Their efficient operation is crucial for ensuring uninterrupted flow in these systems. However, pumps are susceptible to numerous issues that can compromise their performance and reliability. Among these, cavitation stands out as a particularly detrimental phenomenon that can lead to significant damage and operational inefficiencies [1].

Cavitation occurs when the pressure in the pump falls below the liquid's vapor pressure, leading to the formation of vapor bubbles [2]. These bubbles collapse violently when carried to regions of higher pressure, causing shock waves that can erode the pump's components and degrade its performance. The consequences of cavitation extend beyond repair costs, as it can also cause system downtime and energy loss.

While much effort has been devoted to detecting and predicting cavitation in pumps, most of the methods are based on numerical analysis and signal processing techniques [3, 4, 5, 6, 7, 8]. These methods often require extensive domain knowledge and typically work best in controlled environments, where the pump's operating conditions are well understood. However, in real-world scenarios, pumps operate under varying conditions, making it challenging to predict cavitation accurately. This is where ML models can help. By training on data collected from pumps under different conditions, these models can

learn the patterns associated with cavitation and predict its occurrence with high accuracy. ML models are more adaptable, can handle the complexity of real-world data, and can generalize well to unseen conditions.

With the spread of ML techniques, many researchers have started to explore data-driven approaches to predict cavitation in pumps [9, 10, 11, 12, 13]. These approaches have shown very promising results on the efficacy of ML, but their key limitation is that the cavitation detection is done offline.

Different to the related work, we opt for the approach of deploying these ML models directly on embedded systems integrated into pumps. This methodology offers several key advantages. First, it preserves privacy by processing data locally. Second, it eliminates reliance on internet connectivity, which can be inconsistent or unavailable in many industrial settings. Furthermore, processing data on the device leads to lower energy consumption compared to sending data to be processed elsewhere, conserving energy and reducing heat generation. This *Embedded AI* approach necessitates attention to the model's memory requirement and performance, which we address in our study.

In summary, this paper presents a study of a cavitation dataset compiled by the Danish pump manufacturer Grundfos (hereafter referred to as the Grundfos Cavitation dataset), which is unique in size and quality. Specifically, we focus on the problem of pump cavitation detection both as a binary classification problem (i.e. detection of the presence of cavitation) and as a regression problem (i.e. prediction of the amount of cavitation). Aiming to detect the cavitation on pumps, we design models based on SVMs and Deep Neural Networks, and we explore various optimization techniques, including feature optimization and quantization, that reduce the amount of required processing resources. Our experimental results demonstrate that both SVMs and neural networks can provide accurate predictions, with the choice between them depending on the application's specific needs; SVMs offer speed and efficiency, whereas neural networks provide broader functionalities.

2.3 Dataset and Problem Formulation

The Grundfos Cavitation dataset is the result of Grundfos' effort in testing the company's centrifugal pumps in laboratory conditions. To the best of our knowledge, this dataset is unique in size, number of pumps tested, and quality of data. It consists of vibration data from 297 recordings (or experiments) on seven pumps. In each experiment, the corresponding values of flow, pressure, and temperature were captured along with a 30 second long recording of the vibration data sampled at 48 kHz. The settings are kept fixed in each recording but changed between recordings. The pumps contain several stages and are installed vertically. The vibration sensor is located at the top of the pump, while the cavitation is happening at the bottom. This escalates the impact of the pump's stages.

The Net Positive Suction Head (NPSH) is an important concept in the operation of centrifugal pumps. NPSH is used to measure the absolute pressure available in the pump's suction line at the pump impeller's inlet relative to the vapor pressure of the pumped liquid at the operating temperature. It is a measure of how close the fluid at a given point is to boiling, and it is used to ensure that the pump can operate without causing cavitation. The NPSH curve depends on inlet pressure, flow rate, media temperature, inlet pipe diameter, and a few other minor parameters.

NPSH required (NPSHr) is the minimum pressure required at the suction port of the pump to keep the pump from cavitating. The NPSHr is determined by the pump design and operating speed, and it increases with the flow rate. At any point, the pump has an actual

pressure available at the pump suction, called NPSH available (NPSHa), which is a property of the system in which the pump is installed and varies with the system configuration and operating conditions [2]. NPSHa should be greater than NPSHr to prevent cavitation, but a safety margin is usually considered, called NPSHm, and the pump should operate above this margin. NPSHm is described in (2.1). As also seen in (2.1), we defined the Cavitation Factor (CF) as the ratio of NPSHa to NPSHm. Values of CF below one indicate that the pump is cavitating.

$$\mathtt{CF} = \frac{\mathtt{NPSHa}}{\mathtt{NPSHm}}, \quad \mathtt{NPSHm} = \mathsf{max}(\mathtt{NPSHr} + 0.5, 1.2 \cdot \mathtt{NPSHr}) \tag{2.1}$$

By knowing the properties and NPSHr curve of each pump and controlling the flow rate and the inlet pressure, the engineers can estimate the NPSHa and derive the CF. Since the flow and pressure sensors are very expensive, it is preferred to detect cavitation based on the vibration data. In each experiment, the flow and pressure are kept fixed, and the voltage of the vibration sensor is logged. The voltage is then converted to acceleration, and in this study, we further compute its Fast Fourier Transform (FFT). The FFT of the acceleration data is the main feature of the dataset and is used to predict the CF.

Fig. 2.1 illustrates the distribution of recordings and the CF values in the dataset. Fig. 2.2 shows some examples of the FFT of the acceleration data and the corresponding CF values. The figure suggests that, to a certain extent, it might be possible for us as humans to predict the CF value using the FFT information. However, this prediction is not straightforward or highly accurate, as the behavior appears similar across both large and small CF values. The reason for this is that the pumps stall due to the little inlet-pressure in the lab setting. Additionally, certain pumps exhibit unique behavior in specific instances, further complicating the prediction process. These observations suggest that a ML model can be beneficial in predicting the CF based on the FFT of the acceleration, and also might give us good insights if we intend to exploit useful features from the data.

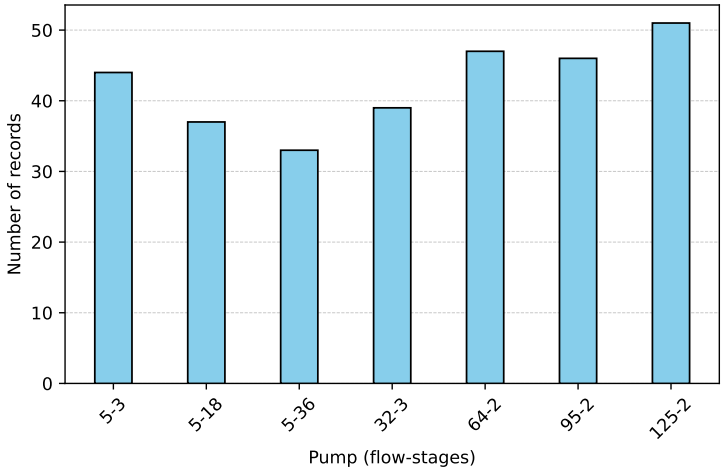
Various problems can be formulated based on this dataset. First, we characterize the problem as a binary classification that has its own use cases and might be relevant when the utilized algorithm only supports classification problems. Then, we change our approach and define the problem as regression. In addition to simple regression, we define a more competent regression problem to focus on the values close to one, and we'll show that this approach can obtain better results even if we put the trained model in a classification situation.

2.4 Methodology

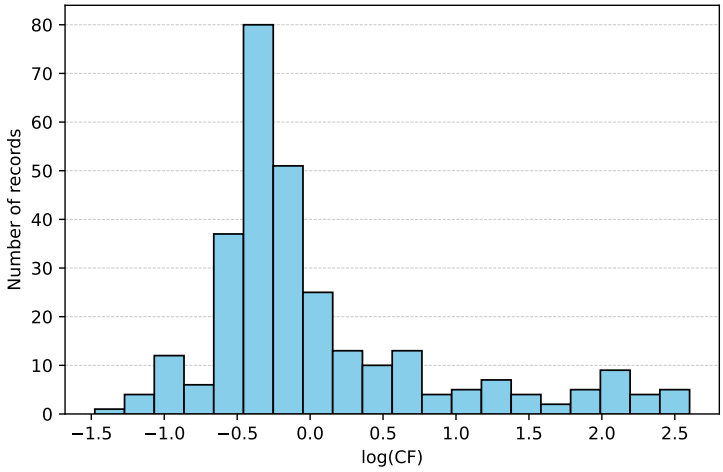
In this study, we used Support Vector Machine (SVM) and deep learning models to solve the Grundfos Cavitation dataset. We considered the support vector machine as a traditional machine learning algorithm that is computationally lightweight compared to deep learning models and still powerful enough to solve many real-world problems. Furthermore, several deep learning models were utilized in this study to compare with the performance of SVMs and benefit from their advantages. The codebase of this study is publicly available [14].

2.4.1 Support Vector Machine

A support vector machine is a powerful and versatile supervised machine learning algorithm used for both classification and regression tasks. It is particularly well-suited for the classification of complex but small- or medium-sized datasets [15]. The basic format of SVM can do linear classification by finding the hyperplane that best separates the classes. However, in many real-world problems, the data is not linearly separable. In such cases,



(a) Number of records for each pump



(b) Distribution of CFs

Figure 2.1: Distribution of dataset elements among different pumps and different values of CF.

SVM uses a kernel trick to transform the input space to a higher dimensional space where the data can be linearly separated. The most common kernels used in SVM are linear, polynomial, and radial basis function (RBF) kernels. In addition to linear kernel, we tested polynomial and RBF kernels, but the performance degraded significantly. Therefore, we only report the results of the linear kernel in this paper.

Each pump has its own attributes, and its vibration data can be completely different from that of other pumps, even after normalization. Therefore, employing a single SVM model for all pumps might not be the best approach. To address this issue, we trained a separate SVM model for each pump and tested the performance of each model. The reporting value will be the average performance of these models. To make it more concrete, we also trained a single SVM model for all pumps and have seen the test accuracy drop from 91.76% to 80.96%.

In the simplest format, the input features are the vibration data captured sequentially in

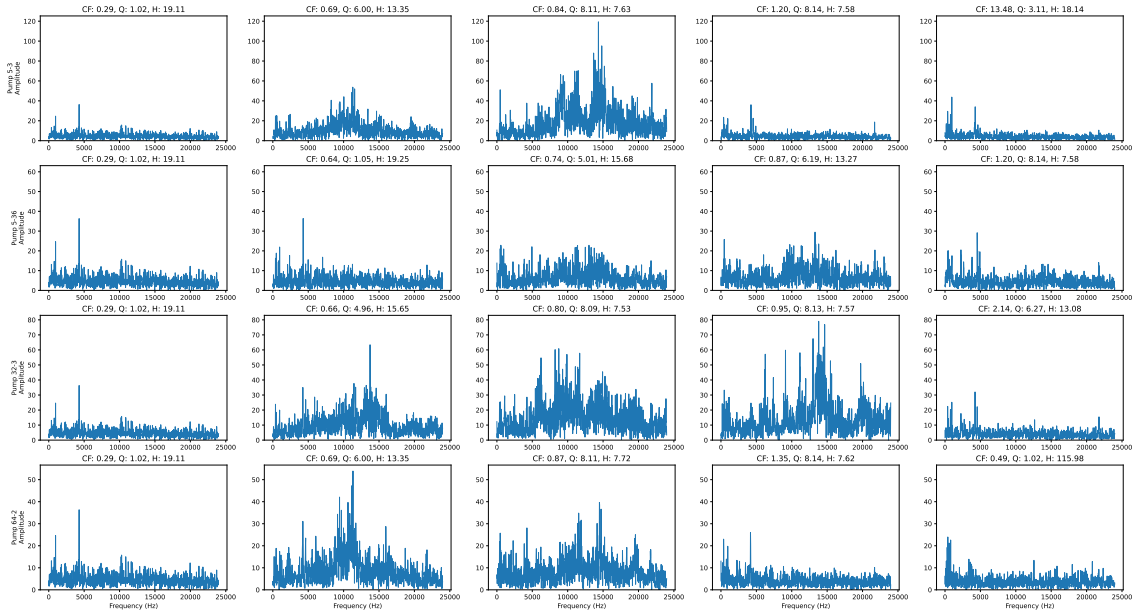


Figure 2.2: Examples of the data. Each row belongs to a specific pump. In the header of each plot, CF is the cavitation factor, Q is the flow (m^3/h), and H is the fluid-independent pressure (m). From left to right in each row, the CF is increasing, i.e., we are going from severe cavitation situations to cavitation free situations, $\texttt{CF} = 1$ being the border of cavitating.

time. Given the high sampling frequency of vibration sensors, the resulting input data can become extensive, leading to increased computational costs and energy consumption for the SVM model. This can be addressed by downsampling the data to the point that the model is still doing well [16]. Furthermore, the feature space constructed from the vibration data may not be well-suited for linear separation. These challenges suggest a need for a feature engineering step to extract the important features from the raw data. In this regard, based on the analysis discussed in Section 2.5, several properties (90th percentile, energy, and standard deviation) of the whole or parts of data were extracted and used to train the SVM models. This approach showed slightly better performance, indicating a better feature space for linear separation, and also can reduce the feature size significantly.

2.4.2 Deep Learning

Deep learning models are capable of automatically discovering the representations needed for feature detection or classification from raw data. This eliminates the need for manual feature extraction, which is one of the key advantages of deep learning over traditional machine learning techniques. It is also capable of solving much more complex problems. For example, in our case, instead of training a separate model on each pump, we were able to train a single model on all of them and reach the same level of accuracy on individual pumps. Still, in most cases, these advantages come with a cost of higher computational complexity, memory usage, and energy consumption compared to traditional machine learning.

Since the raw data we'll feed to the deep learning models is large, naively employing fully connected layers would lead to an excessively high number of parameters, which is undesirable. Therefore, we used Convolutional Neural Networks (CNN), which are well-suited for processing the spatial structure in the data and fit well with the sequential

nature of the vibration data. Recurrent Neural Networks (RNNs), including LSTMs and GRUs, were not deployed because of their complexity and the fact that the data does not exhibit long-term informational diversity. This makes the use of CNNs with proper windowing more beneficial. Initially, we defined the regression problem in its basic form, i.e., predicting the $\texttt{CF}$ value. Respecting the properties of the dataset and keeping in mind the importance of the values close to one, we defined a better regression problem to focus on these values. First, the $\texttt{CF}$ value ranges between zero and positive infinity with a cavitating threshold of one. We defined a new target value as $\log(\texttt{CF})$ to make the problem more balanced and easier for the model to learn. Second, as we get away from the threshold, the change in the pattern of the vibration data becomes subtle and will very soon become negligible. For example, even the difference between $\texttt{CF} = 4$ and $\texttt{CF} = 5$ is less significant than the difference between $\texttt{CF} = 1.1$ and $\texttt{CF} = 1.2$. Considering the usual applications of this system (like monitoring the condition of the pump and taking preventive actions), it is much more critical to accurately predict the $\texttt{CF}$ values close to one and the difference between $\texttt{CF} = 4$ and $\texttt{CF} = 5$ can even be ignored. Therefore, we defined our final target value as $\sigma(\alpha \times \log(\texttt{CF}))$, in which σ represents the sigmoid function, and α is a hyperparameter affecting the range of values close to the border that the model should be sensitive to. In our experiments, we have found that $\alpha = 4$ is a good choice.

2.5 Results

We conducted many experiments to find the best combination of settings under which the models could perform well. The results of these experiments are reported in this section.

2.5.1 Support Vector Machine

Our default setting and problem formulation are as follows: Given the suitability of SVMs for classification tasks, we have framed our problem as a binary classification challenge. The data is split by a window size of 4096 samples; and 20% of the data is used for testing, and the rest is used for training. The train set and test set are kept distinct by ensuring that no part of the test data comes from the same recording as any of the training data. For each data, we computed the 90th percentile, energy, and standard deviation of the entire signal and subsequently for each of the 50 subdivisions of its length, yielding a total of 153 properties, which were then considered as input features for our model.

All experiments were performed three times, and our observations and conclusions remained consistent throughout.

Data Inclusion

We considered feeding the model with the FFT of the acceleration data, 90th percentile, energy, and standard deviation of the whole data with or without subdivisions. This is due to the fact that, unlike deep neural networks, SVMs are not capable of automatically discovering the representations needed for classification, and preparing the input data is crucial for the performance of the model. The selection of these feature spaces was based on their rich representation and, in most cases, their low cost of extraction. This decision was partially inspired by [17].

Table 2.1 shows the performance of the SVM model receiving various input features. It proves that the 90th percentile (90th), energy, and standard deviation (SD) can all be good features for the model, energy being the best. However, a combination of these features can improve the model's performance. Remarkably, this combination outperforms the FFT of the data and achieves equivalent performance when integrated with the FFT. This is crucial because the FFT generates a lengthy sequence, significantly increasing the model's computational burden, which can be avoided. This computational burden

is related to the number of multiplications and accumulations (MACs), as detailed in the table.

Table 2.1: Including combination of data types

	Accuracy (%)	MACs
FFT	91.21	4096
90th	89.89	306
Energy	91.54	306
SD	90.77	306
90th, Energy, SD	91.76	918
FFT, 90th, Energy, SD	91.75	5014

Another variable that can change the performance of the model is the number of subdivisions for which we compute statistics. Increasing these numbers should give more detailed information about the behavior of the signal and should improve the performance of the model in the cost of computational complexity. However, we expect this improvement in accuracy to saturate. Finding a good number of subdivisions is crucial to balance the performance and computational cost of the model.

In Table 2.2, we show the performance of the SVM model with different numbers of subdivisions. Based on it, the importance of using subdivisions for giving the model more detailed and spatial information about the signal becomes clear (*0 parts* versus *5 parts*). Further, up to *50 parts*, we see a good improvement in the performance, but there is not much improvement after that. Therefore, we chose *50 parts* as a good number of subdivisions for our model.

Table 2.2: Partitioning strategies on SVM model

Parts	**0**	**5**	**10**	**25**	**50**	**100**
Accuracy (%)	73.06	83.75	87.97	89.92	91.76	91.95
MACs	6	36	66	156	306	606

Window Size

Each data point in the original dataset belongs to a separate test (also called recording), consisting of 30 seconds of continuously logging the vibration data at 48 kHz. This obviously is a very long sequence and is mostly repetitive. Therefore, we windowed the data into smaller sequences and fed them to the model. Although not much relevant to the SVM model, in case of deep learning models, different windows of the same recording will be similar to each other and can be considered as augmented data, preventing the model from overfitting. We've found a window size of 4096 samples to be a consistent good choice through several experiments, Table 2.3 illustrating one of them. It can be concluded that a data frame of around 85 milliseconds with a high enough sampling rate should contain enough information for successfully detecting the cavitation.

Table 2.3: Division of data by different window sizes

Window Size	**256**	**1024**	**4096**	**16384**	**65536**	**262144**
Accuracy (%)	82.12	88.38	91.76	91.44	90.84	92.18

Other Experiments

As discussed before, training a separate SVM for each pump should get better results than having a single SVM for all pumps. To prove this, we compared the performance of

these two approaches. If a model is trained on each pump separately, an average test accuracy of 91.76% is achieved, while this value degrades to 80.96% if a single model is trained for all pumps.

We also tested the performance of the SVM model with different kernels. Using the linear kernel, 91.76% of the data will be classified correctly. However, using the polynomial and RBF kernels, this value goes down to 75.52% and 82.16%, respectively. This is a clear indication that the data is linearly separable, and among these, the linear kernel is the best choice for this problem.

2.5.2 Deep Learning

The same default setting as the SVM model was used for the deep learning model, except that we fed the model with the FFT of the acceleration data and a single model was trained on the whole dataset of pumps. The problem is also formulated as a regression problem, and the target value is defined as $\sigma(4 \times \log(\mathtt{CF}))$. To make the results being more sensible, instead of reporting the loss value, mean squared error (MSE), or mean absolute error (MAE), we check whether the prediction of the model and the actual value both indicate the cavitation situation and report the accuracy of the model in the assumed classification problem.

We have studied various network architectures, the performance of the network in classification and regression setups, and the impact of meticulously choosing the target value. Moreover, a spectrum of different sizes of the model was benchmarked on the STM32L4 microcontroller using TensorFlow Lite for Microcontrollers (TFLM) [18], a necessary step to study the impact of models in real-world applications that is missing from most of the literature in this field.

Model Architecture

We tested several deep learning architectures. More specifically, a fully connected network, a convolutional neural network without global average pooling, and several sizes of convolutional neural networks with global average pooling were studied. All models used batch normalization layers and they used *ReLU* as the activation function. They were trained for ten epochs with a batch size of 32.

Each neuron in the first layer of the fully connected network will increase the static memory requirement of the model by more than 8 kB. This is significant, and therefore, we only used 16 neurons in the first layer. Thereafter, a layer of 10 neurons connected these features to the single output neuron. The model achieved an accuracy of 90.41% on the test set.

Convolutional layers can find the properties of the data by sliding a kernel over it, making them parameter-efficient and suitable for processing the spatial structure in the data. Based on the properties of the data, we chose relatively large kernels. In our default model architecture, we used a global average pooling layer at the end of convolutional layers to conclude the spatial information and significantly reduce the number of neurons in that feature map. This model achieved an accuracy of 90.60%, in par with the fully connected network, but having about three times less parameters. This is why we preferred this model over the fully connected network, although it's important to note that the convolutional model requires more computations. In certain scenarios, the fully connected model might be more suitable.

With the intuition that the global average pooling layer removes the spatial information, we tested the performance of the convolutional model without this layer. The model achieved

an accuracy of 91.03%, which, as expected, is slightly better than the base model. Meanwhile, this modification increased the number of parameters by 67%, which is undesirable.

An experiment for comparing the classification and regression setups was conducted in which the same base model as in regression was used in the classification problem. The model achieved an accuracy of 89.27% in the classification setup, which is worse than 90.60% achieved in the regression setup. This is a clear indication that the regression setup is more suitable for this problem.

Instead of using the value of CF for regression, we defined a new target as $\sigma(4 \times \log(\mathtt{CF}))$ to make the problem more balanced and easier for the model to learn. To prove the importance of this approach, we tested the model's performance with the original target value. The simple regression model achieved an accuracy of 88.07%, while the model with the new target achieved an accuracy of 90.60%. Although the superiority of the new target can be concluded from its accuracy, the difference is more significant when we look at their prediction graph in Fig. 2.3. The model trained by the new target can more accurately follow the sigmoid function shape that we expected it to learn. Another observation of this figure can be that some data points close to the border might be identified as misclassification, even though the model has a good prediction for them.

Based on the default CNN model that we've shown to be successful, we designed several smaller models. Table 2.4 contains information of these models and their performance. *CNN_4* is the default CNN model.

Table 2.4: CNN models' information

	Params	**MACs**	**Loss**	**Accuracy (%)**
CNN_1	1,089	557,592	0.0858	75.11
CNN_2	3,409	2,925,080	0.0580	80.68
CNN_3	8,561	2,114,072	0.0333	87.77
CNN_4	11,409	940,632	0.0238	90.60

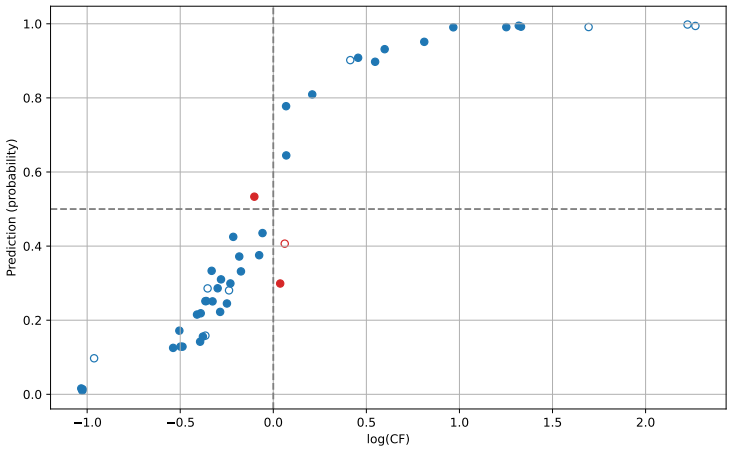
We have deployed the models to a NUCLEO-L4R5ZI board and benchmarked their performance. NUCLEO-L4R5ZI contains an STM32L4R5ZI microcontroller, which is based on an Arm Cortex-M4 core. TFLM is used to port the model to its C++ implementation. In addition to the float32 format, several quantized versions of the models have been tested, namely, dynamic range quantization, int8 quantization, 16x8 quantization, and float16 quantization. Since the most effective quantization method was the int8 quantization, we only report the results of this method and the default float32 format. Table 2.5 contains the benchmarking results of the models.

Table 2.5: CNN models' performance

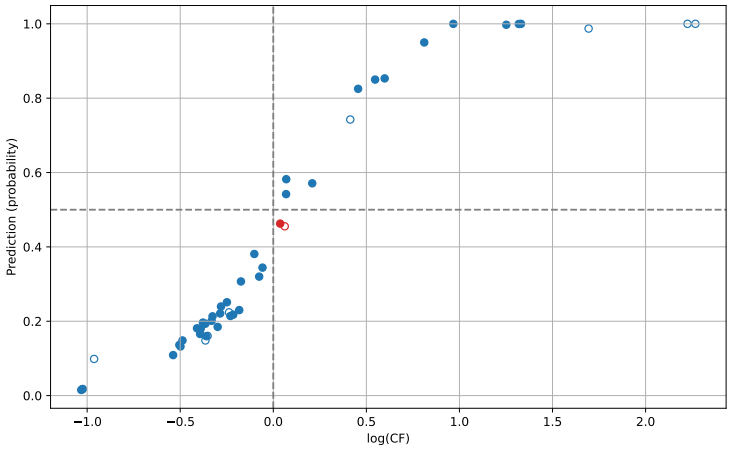
	float32			**int8**		
	Exe time	**Flash**	**RAM**	**Exe time**	**Flash**	**RAM**
CNN_1	3159.06	420.7	221.5	825.19	**358.8**	61.6
CNN_2	13362.05	430.3	279.8	2449.60	362.1	75.3
CNN_3	9442.18	450.9	150.9	1594.58	368.5	44.0
CNN_4	4354.99	463.1	127.5	**731.78**	373.3	**38.1**

Exe time (execution time) is in ms. $Flash$ and RAM are in kB.

This table reveals that the number of parameters is not the only factor constructing the memory requirement, but there are other requirements of TFLM that are more significant.



(a) CF as target



(b) $\sigma(4 \times \log(\mathtt{CF}))$ as target

Figure 2.3: Impact of defining various regression targets for the model. Hollow circles represent data present in the test set, while filled circles denote data in the training set. Red indicates misclassification, whereas blue signifies correct classification.

For example, the weights of *CNN_4* will take around 45 kB of the flash memory in the float32 form, but still around 418 kB more is needed for the model to be deployed (a big chunk of it is required for TFLM operations registration and depends on the model we are deploying).

In a separate experiment, we found that the floating point multiplication and integer multiplication in STM32L4R5ZI take almost the same time (this microcontroller has an FPU). The speedup of the int8 quantization is due to the optimized implementation of integer operational kernels in CMSIS-NN library [19] that is used in conjunction with TFLM. The main portion of the speedup is due to the utilization of the Arm Cortex-M4 core's SIMD instructions. When comparing the execution time of *CNN_1* and *CNN_4*, it can be observed that *CNN_1* performs faster in the float32 format, while *CNN_4* excels in the int8 format. This underscores the significance of such comparisons in practical applications, invalidating many of our presumptions.

Since the int8 model improves all resource benchmarks without compromising the ac-

curacy of the model, we can conclude that the int8 quantization of *CNN_4* is probably the best choice for this problem, showing advantages in all aspects of the benchmarking, except for the flash memory requirement which was in par with the other int8 models.

In comparing SVMs with neural networks, it is evident from Tables 2.1, 2.2, and 2.4 that SVMs require orders of magnitude less computation than neural networks. While this efficiency is a notable advantage, it comes at the cost of SVMs' limited functionality compared to neural networks. Consequently, the decision to opt for one over the other should be guided by the specific application requirements.

2.6 Conclusion

This study illustrated the potential of machine learning techniques, notably support vector machines and deep learning models, in detecting pump cavitation. By leveraging the dataset provided by Grundfos, the study offered compelling evidence that machine learning approaches can accurately predict cavitation occurrences in pumps under varying real-world conditions. The study's methodology, which included detailed data analysis, feature engineering, problem formulation, model design, and rigorous testing, provided valuable insights into the challenges and opportunities in this domain.

The results of the study demonstrated that deep learning models, particularly convolutional neural networks, can effectively learn the patterns associated with cavitation and predict its occurrence with high accuracy. The study further emphasized the advantages and constraints of SVM models in this context. Particularly, with effective feature engineering, SVM models have the potential to surpass the performance of more intricate deep learning models in specific situations. However, this comes at the cost of reduced generality and limitation to classification problems. The study's findings underscored the importance of testing machine learning models on target hardware to ensure their practical applicability, a missing point in most of the embedded artificial intelligence literature, especially in the context of industrial applications.

Although a very compact and powerful model was designed in this study, still it is interesting to see how the model can be further optimized by leveraging the Neural Architecture Search (NAS) and model compression techniques [20, 21, 22, 23, 24, 25], especially those designed for embedded systems [16, 26, 27]. This can be a promising direction for future research.

3 Paper B: EdgeMark: An automation and benchmarking system for embedded artificial intelligence tools

3.1 Paper Abstract

The integration of Artificial Intelligence (AI) into embedded devices, a paradigm known as embedded artificial intelligence (eAI) or tiny machine learning (TinyML), is transforming industries by enabling intelligent data processing at the edge. However, the many tools available in this domain leave researchers and developers wondering which one is best suited to their needs. This paper provides a review of existing eAI tools, highlighting their features, trade-offs, and limitations. Additionally, we introduce EdgeMark, an open-source automation system designed to streamline the workflow for deploying and benchmarking ML models on embedded platforms. EdgeMark simplifies model generation, optimization, conversion, and deployment while promoting modularity, reproducibility, and scalability. Experimental benchmarking results showcase the performance of widely used eAI tools, including TensorFlow Lite Micro (TFLM), Edge Impulse, Ekkono, and Renesas eAI Translator, across a wide range of models, revealing insights into their relative strengths and weaknesses. The findings provide guidance for researchers and developers in selecting the most suitable tools for specific application requirements, while EdgeMark lowers the barriers to adoption of eAI technologies.

3.2 Research Highlights

- Review of embedded AI tools, their features, strengths, and limitations
- EdgeMark: an open-source edge AI automation and benchmarking system
- TensorFlow Lite Micro (TFLM) versus Edge Impulse, Ekkono, and Renesas eAI Translator

3.3 Keywords

Machine learning; TinyML; Embedded AI; Automation; Benchmarking; Microcontrollers

MSC 68T99

3.4 Introduction

AI is revolutionizing how systems process, analyze, and act on data. While historically rooted in cloud computing, the need for faster, more secure, and localized AI processing has driven the rise of eAI. Unlike conventional AI, eAI processes data at the device level, transforming devices into standalone intelligent systems. This shift enables a wide range of applications, from smart home devices and wearables to industrial automation and healthcare [28].

However, embedding AI into resource-constrained devices introduces challenges, including processing power limitations, memory constraints, and energy efficiency requirements [29]. To address these challenges, a variety of tools have been developed to facilitate the

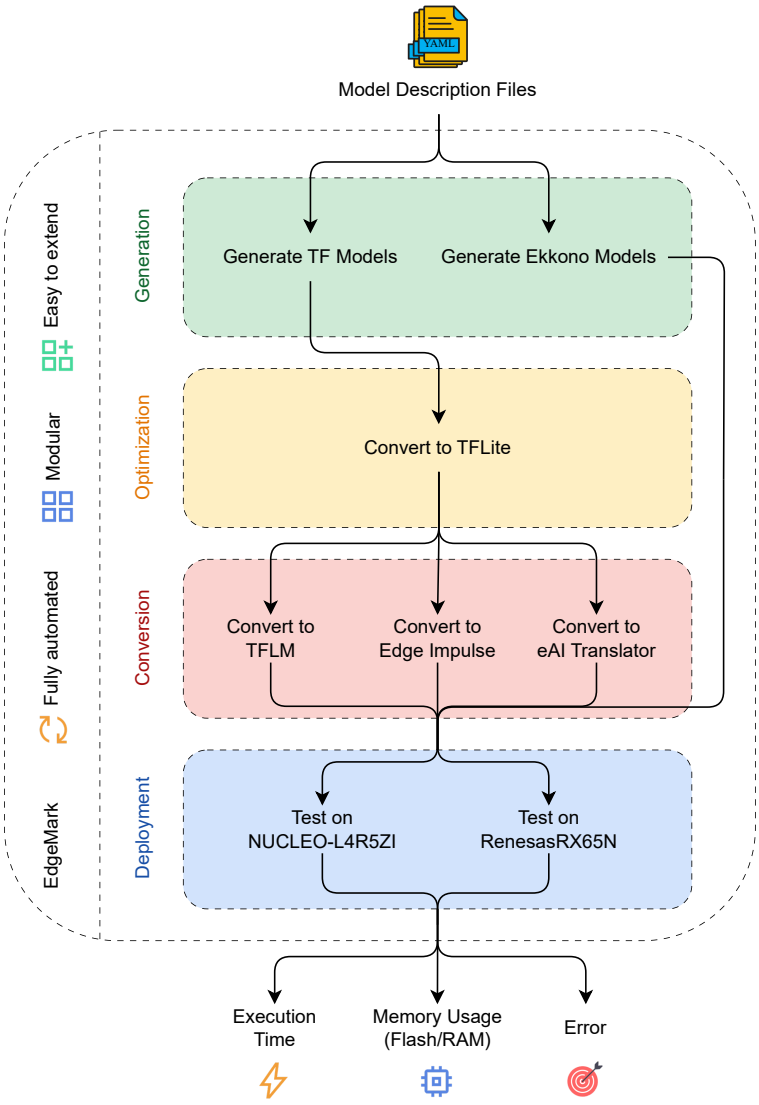


Figure 3.1: Graphical abstract.

deployment of ML models on embedded devices. While these tools simplify the development process, they exhibit significant trade-offs in functionality, ease of use, and performance across different platforms. A concise review and benchmarking of these tools is necessary to guide developers in making informed decisions.

Beyond individual tools, deploying ML models on embedded devices typically involves a multi-stage workflow comprising model generation, optimization, conversion, and deployment. Completing these steps, often manually, can be time-intensive and error-prone, particularly when scaling across multiple models or iterations. Automation systems designed to streamline this workflow have the potential to significantly reduce development time and improve consistency. To this end, we developed EdgeMark, a system that automates the entire workflow by simply taking in details about the user's model and their choice of tools and target hardware. It acts as an orchestrator that abstracts away tool-specific complexity and efficiently coordinates their use.

Accordingly, the contributions of this paper are threefold:

1. **Review of eAI Tools:** We provide an overview of existing eAI frameworks, outlining their supported features, strengths, and limitations.
2. **EdgeMark Automation System:** We introduce EdgeMark, an open-source framework designed to automate the deployment and benchmarking workflow, simplifying model generation, optimization, conversion, and deployment on embedded platforms. The codebase for EdgeMark is available here: https://github.com/Black3rror/EdgeMark.
3. **Benchmarking eAI Tools:** Through extensive experiments, we evaluate the performance of widely used eAI tools across a wide array of models, highlighting trade-offs and differences they exhibit in various scenarios.

Through the combination of these contributions, we aim to not only contextualize the current capabilities of eAI tools but also provide practical guidance and help for deploying ML models on embedded systems. By facilitating modularity, scalability, and reproducibility, EdgeMark represents an important step toward addressing the challenges of eAI deployment and paving the way for the next wave of intelligent applications at the edge.

The remainder of this paper is organized as follows. Section 3.5 provides an overview of related works. Section 3.6 discusses the key features, functionality, and limitations of eAI tools. In Section 3.7, we describe the automation system developed to streamline the use of these tools, including its architecture and use cases. Section 3.8 outlines the benchmarking methodology and presents experimental results comparing the performance of the tools across multiple parameters. Finally, Section 3.9 concludes the paper by summarizing the contributions and proposing directions for future work.

3.5 Related Work

Review of eAI tools: While many TinyML surveys are not specifically focused on eAI tools, they often include these tools as part of their review [28, 29, 30, 31, 32, 33, 34, 35]. These surveys typically provide a brief overview of available tools, covering many important ones but lacking completeness. Given the rapid evolution of the field, some information happens to be missing or becomes outdated. In response, our work presents a more thorough and current review of ten eAI tools, highlighting their features, advantages, and limitations.

Benchmark of eAI tools: MLPerf Tiny [36] is a benchmark suite for TinyML models,

designed to provide a standardized method for evaluating the performance of TinyML models across different platforms. It includes four benchmarks in widely used TinyML applications. Although it has played a significant role in improving TinyML tools, we identify two main limitations in its use for comparing eAI tools. First, it includes only four models, all of which are considered large for many embedded devices. As a result, it cannot provide a comprehensive view of the tools' performance across a wide range of models, especially given that many industrial applications of eAI involve small models requiring less than 100 kB of memory. Second, MLPerf Tiny does not impose fixed hardware or configuration settings, which makes it challenging to compare the performance of different tools, as each may be evaluated under varying test conditions.

Several studies have compared popular eAI tools. For instance, [37] examines the performance of TFLM and STM32Cube.AI on a few of models, concluding that STM32Cube.AI outperforms TFLM but is limited to STM32 devices, whereas TFLM offers broader flexibility. In [38], the creators of AIfES compare its performance against TFLM, showing that AIfES outperforms TFLM, particularly on fully connected (FC) models. Similarly, [39] contrasts the interpreter and non-interpreter versions of TFLM with microTVM. This work also introduces a third backend for Tensor Virtual Machine (TVM) that optimizes RAM usage through unified static memory planner (USMP) and runtime improvements. Although these studies provide valuable insights, they generally focus on comparing two tools within constrained test environments, falling short of offering a comprehensive evaluation.

In contrast, EdgeMark offers a more extensive and systematic evaluation of eAI tools. By managing the entire workflow, it ensures consistent testing conditions, allowing for fair comparisons between tools. This enables benchmarking across a diverse set of models, from very small to relatively large, and across multiple tools. Benchmarking results are discussed in Section 3.8.

Automation of eAI tools: Few works address the automation of benchmarking systems for TinyML. In [39], the authors implement automation to a certain extent, allowing them to conduct numerous comparisons within a short timeframe. [40] adopts this principle at its core but focuses more on traditional ML models and tools. Nevertheless, the need for a comprehensive automation system capable of handling the entire workflow for deploying ML models on embedded devices remains evident. To address this gap, EdgeMark is designed as a modular and scalable solution that automates both deployment and benchmarking of eAI tools.

3.6 eAI Tools

AI models are commonly represented in high-level formats such as TensorFlow, PyTorch, or ONNX. While these formats facilitate model development, they are not always ideal for deployment on resource-constrained devices. This is because they are designed with rich interpreters in mind and include numerous features that embedded systems might not support efficiently. Therefore, AI models must be converted to formats that are optimized for execution on the target device. Additionally, there may be opportunities for optimizing model execution specific to the hardware.

A variety of tools exist for deploying AI models on embedded devices. These tools take advantage of their understanding of the target device's architecture to optimize model execution. Due to the diverse range of hardware available, different tools are suited for different devices. For instance, STM32Cube.AI is widely used for deploying models on STM32 devices, whereas the eAI Translator is tailored to match the specific hardware features of Renesas microcontrollers.

These deployment tools typically follow one of two approaches: direct execution or interpreter-based execution. Some tools incorporate the model execution directly into the firmware, whereas others convert the model into a binary format and use a separate interpreter to execute it. The interpreter approach tends to offer faster and more flexible development, enabling features such as shared memory management for multiple models and the ability to update models without altering the firmware. In contrast, direct execution can be more efficient in terms of both execution speed and memory usage.

Beyond model conversion, some tool providers offer comprehensive solutions that include data collection and labeling, model generation and testing, and data preprocessing and postprocessing. This broader scope makes the term “toolchain” or “platform” more appropriate for these offerings. In the following subsections, we will discuss some of the most popular tools for eAI.

3.6.1 TensorFlow Lite

TensorFlow Lite (TFLite), recently renamed LiteRT, is a lightweight, fast, and optimized framework designed specifically for deploying DL models on mobile and embedded devices. It converts models into FlatBuffers, a compact and efficient binary format that minimizes overhead. During conversion, TFLite can apply various model optimization techniques, such as quantization, pruning, and clustering. The resulting model can be executed on devices using the TFLite interpreter or used as input for further optimization or conversion to the target device’s native language.

While TFLite primarily targets mobile devices and more capable embedded systems, TFLM [41], developed by Google in 2018 as an open-source project, is specifically tailored for microcontrollers and other resource-constrained devices with only a few kilobytes of memory. Although the heterogeneity of the devices was mentioned as a reason for having many tools, TFLM carries the ambition to unify them under a single framework by following a general kernel-based approach, whereby each vendor can optimize the kernels for their specific hardware. This capability has led to TFLM’s widespread acceptance among researchers and developers. For instance, CMSIS-NN provides optimized kernels for ARM Cortex-M processors and is integrated into TFLM. Additionally, TFLM includes optimized implementations for ARM Ethos-U processor series, Cadence Xtensa processor architecture, CEVA neural processing units, and ARC processors.

TFLM processes a TFLite model and converts it into C++ source code that can be compiled with any C++17-compliant compiler. It operates without requiring an operating system or dynamic memory allocation, with the bare interpreter using less than 2 kB of memory [41]. Although the original design of TFLM utilizes an interpreter to execute the model, there have been few attempts for direct model execution on devices. [39] demonstrates the success of one of these works in lowering the memory usage and significantly improving the setup time for the model, although the execution time remains the same.

3.6.2 Edge Impulse

Edge Impulse [42], founded in 2019, is a cloud-based platform that offers a comprehensive toolchain for developing, training, and deploying ML models on embedded devices. It has evolved to support a wide array of devices and sensors, offering both free and enhanced professional and enterprise versions with additional features.

One of the key features of Edge Impulse is its flexibility with datasets. Having a dataset in their platform allows the users to find and train a model that fits their data best. Users can upload their own datasets or connect a device directly to the platform to collect data in real time. The platform simplifies data labeling and also supports data augmentation to

expand the dataset and improve model generalization.

For data preprocessing, Edge Impulse offers a variety of signal processing blocks, paving the way for efficient model training. Users can choose from a range of ML algorithms, including neural networks [43], logistic regression, SVM, random forest, extreme gradient boosting (XGBoost), and light gradient boosting machine (LightGBM). The platform's EON Compiler stands out as an interpreter-less code generator for TFLM, reducing memory usage while maintaining kernel efficacy [44]. Recently, the EON Compiler has provided enterprise users with the option to trade off between memory usage and execution time. To achieve better performance, Edge Impulse supports 8-bit quantization.

Deployment flexibility is another strength of Edge Impulse. Users can output C++ source code for compilation on most embedded devices or generate a library for a specific integrated development environment (IDE) or a binary for direct device flashing. The platform supports an extensive range of hardware, from microcontrollers like the Arduino Nano 33 BLE and ESP32 to more powerful devices such as the Raspberry Pi and even Nvidia Jetson Nano. Additionally, Edge Impulse provides an estimation of the model's execution time, flash size, and RAM usage on the target device.

3.6.3 microTVM

microTVM [45] is a part of the Apache TVM project, which is an open-source DL compiler stack used to optimize and deploy ML models on a variety of hardware platforms. microTVM is specifically designed to support deployment on microcontrollers and other resource-constrained devices.

microTVM uses the TVM stack to compile ML models into highly optimized code. The compilation involves several optimization steps such as operator fusion, memory planning, and hardware-specific optimizations. Binding with AutoTVM, a TVM module for search and tuning the performance of operators on a specific hardware target, allows microTVM to generate highly optimized operators for the target device. For tuning tasks, AutoTVM typically requires some level of interaction with the target device, such as connecting to the microcontroller via USB-JTAG and accessing a cross-compiler and an on-chip debugger, to evaluate the performance of generated schedules.

To further enhance model efficiency, microTVM supports quantization. It does not require an operating system or dynamic memory allocation, making it suitable for bare-metal deployment. Although TFLM uses hand-optimized kernels like CMSIS-NN to ensure maximum efficacy and microTVM automatically generates optimized operators for the specific use case, microTVM is able to achieve comparable performance to TFLM [46].

3.6.4 STM32Cube.AI

STM32Cube.AI [47] is a powerful tool developed by STMicroelectronics in 2018 aimed at simplifying the deployment of neural network models on STM32 microcontrollers. While its source code is proprietary, the tool is freely accessible to STM32 developers. STM32Cube.AI integrates seamlessly with STM32CubeMX and can be accessed through graphical interface, command line interface, or online.

This tool efficiently converts models from popular frameworks such as TensorFlow, Keras, TFLite, PyTorch, Matlab, and Scikit-learn into C code. It employs techniques such as 8-bit or mixed-precision quantization, alongside graph and memory optimizations, to ensure optimal performance. In one of their comparative tests against TFLM, STM32Cube.AI has been shown to achieve, on average, 36% faster execution time, 24% smaller flash size, and 26% reduced RAM usage for image classification and visual wake words models, as part of the MLPerf Tiny benchmark [48].

Beyond its conversion capabilities, STM32Cube.AI offers users a comprehensive model zoo and supports them with numerous tutorials and examples. To aid user comprehension, it provides a visual representation of models and their layers, accompanied by detailed memory usage breakdown for each layer. Another interesting feature of STM32Cube.AI is the possibility of testing the models on real STM32 boards in their board farm. This will give users a measure of the model's latency, accuracy, and memory footprint on various STM32 devices, which can be helpful in choosing the right device for their application. Additionally, STM32Cube.AI supports storing model parameters and activations in external memory, which can be useful for models with large memory requirements.

3.6.5 Renesas eAI Translator

Renesas Electronics Corporation, a prominent Japanese semiconductor manufacturer, introduced the eAI Translator [49] in 2017. This tool allows for the deployment of ML models on Renesas microcontrollers and operates as a plugin for Renesas' IDE for embedded systems development, e2 studio. The eAI Translator generates C code based on models provided by the user, supporting frameworks such as TensorFlow, Keras, TFLite, and PyTorch.

The tool is compatible with a wide range of microcontroller families, including RL78, RX, RA, RZ, and Renesas Synergy. It executes models directly on the microcontroller without requiring an interpreter. Additionally, eAI Translator supports 8-bit quantization with TFLite models and offers a specific version of the CMSIS-NN library, optimized for Renesas RX and RA architectures.

One notable feature of eAI Translator is its user-friendly interface, which simplifies the model conversion process. Users can decide whether to prioritize execution speed or RAM usage, a choice that determines how model parameters are allocated in either RAM or ROM. This flexibility allows users to execute small models even faster by storing them in RAM.

3.6.6 Ekkono

Founded in 2016, Ekkono [50] is a Swedish company that offers a platform for developing and deploying ML models on embedded devices. A unique feature of Ekkono is its on-device learning capability, which sets it apart from many other tools. This feature allows models to adapt to their environment and improve their accuracy over time. While the computational and memory costs of this technique might raise some concerns, its potential to enable new applications at the edge is undeniable.

Unlike most tools that take in pretrained models in formats such as TensorFlow, ONNX, or TFLite, Ekkono requires users to develop and train models directly within its platform. This necessitates learning the Ekkono-specific model development process. Ekkono Primer is their main software library to develop models. Once models are developed, they can be deployed to edge devices using either Ekkono Edge or Ekkono Crystal.

Ekkono Edge is a C++ library designed for devices with a few megabytes of memory, such as communication gateways or programmable logic controllers (PLCs). In contrast, Ekkono Crystal is a C library with limited functionality, optimized for microcontrollers that have only a few kilobytes of memory. Ekkono Crystal only relies on standard C libraries and can be compiled with any C99-compliant compiler. This library only supports linear regression and Multi-Layer Perceptron (MLP) models with Sigmoid, Tanh, and LeakyRelu nonlinearities, targeting regression tasks.

Additionally, Ekkono offers Ekkono Spectral for signal processing needs. The platform also includes features like data preprocessing, automated machine learning (AutoML),

sensitivity analysis, model change detection, conformal prediction, and anomaly detection. Ekkono relies on an interpreter to execute models on the edge. While Ekkono's platform is not free and requires licensing, it is well-documented, and users find it easy to use once they become familiar with its development process.

3.6.7 ARM-NN

Introduced by ARM in 2018, ARM-NN [51] is an open-source project that targets more capable devices operating on Linux or Android. This software development kit (SDK) is specifically designed to optimize performance on hardware architectures like ARM Cortex-A CPUs, ARM Mali GPUs, and ARM Ethos neural processing units (NPUs). It supports popular ML frameworks, including TensorFlow, Caffe, and ONNX, leveraging the Compute Library for efficient hardware acceleration. Similar to CMSIS-NN, ARM-NN aims to enhance the deployment of neural networks across a range of ARM devices.

3.6.8 Embedded Learning Library (ELL)

ELL [52], an open-source project launched by Microsoft Research in 2017, aims to simplify the deployment of ML models on resource-constrained platforms. It is particularly focused on small single-board computers such as the Raspberry Pi, and also supports devices like Arduino and micro:bit.

The project enables users to convert models from Microsoft Cognitive Toolkit (CNTK), Darknet, or ONNX into the ELL format, which serves as an intermediate representation. This format is then transformed into C++ code that can be compiled and executed on the target device. Alongside the C++ capabilities, ELL offers Python bindings for performing model inference.

It's important to note that ELL is currently inactive, and its codebase has recently been archived.

3.6.9 NanoEdge AI Studio

NanoEdge AI Studio [53], an AutoML platform developed by Cartesiam in 2020 and later acquired by STMicroelectronics, streamlines the process of finding optimal ML models for user data. It accepts input data in text or CSV formats and supports various applications, including anomaly detection, outlier detection, classification, and regression tasks. Notably, the platform supports on-device learning for anomaly detection.

The output of the platform is a static library that can be linked to the user's C code and served to any ARM Cortex-M microcontroller. The generated models are lightweight, occupying under 20 kB of RAM and flash memory, and can execute in less than 20 milliseconds on a Cortex-M4 processor operating at 80 MHz [54]. This platform features an intuitive and user-friendly interface and has recently become completely free [55].

3.6.10 uTensor

uTensor [56] is a free and open-source project developed by Arm, designed to convert TensorFlow models into C++ code. This code is primarily targeted at MBed OS, an open-source embedded operating system from Arm. In 2019, Arm announced that the uTensor project would be merged into TFLM [57]. More recently, Arm has also declared that MBed OS has reached its end of life, and it is no longer maintained or supported [58].

3.6.11 Others

We have identified approximately 50 tools related to eAI. In addition to the tools previously mentioned, we list the following: micromind [59], ExecuTorch [60], Imagimob [61], OmniML [62], EdgeML [63], TinyEngine [64], FANN-on-MCU [65], AIfES [66], TinyML gen [67], CMix-NN [68], Neurona [69], hls4ml [70], Silicon Labs MLTK [71], Microsoft NNI [72],

Core ML [73], MNN [74], Glow [75], eIQ [76], ONNX Runtime [77], QKeras [78], Larq [79], ONNC [80], Latent AI [81], Plumerai [82], NNTool [83], and DORY [84]. Additionally, as exemplified in [85], traditional ML algorithms hold significant promise for addressing many industrial applications. As a result, certain eAI tools have been specifically developed to support these algorithms. These include: sklearn-porter [86], weka-porter [87], m2cgen [88], MicroMLGen [89], EmbML [90], Qeexo [91], and emlearn [92].

3.6.12 Summary

Table 3.1 presents a summary of key features found in several prominent eAI tools. It includes details such as the tool's developer, release year, input and output formats, supported models, hardware compatibility, interpretability, licensing, on-device learning capabilities, and ease of use. In the *Model* column, *ML* is a general term that typically includes *DL*. For the *Hardware* column, *kB*, *MB*, or *GB* roughly indicates the RAM size of the target device, while other entries more specifically indicate the target device. The *ease of use* rating reflects the authors' subjective evaluation, which may be based on available tutorials, documentation, demos, or limited hands-on testing, rather than extensive practical experience.

Although most eAI tools are developed within industry, driven by the strong demand for deploying machine learning models in commercial products, some originate in academia. For instance, microTVM began as a research project at the University of Washington [94] before becoming part of the Apache TVM ecosystem, while TinyEngine, developed as part of MIT's MCUNet project, is the product of entirely academic work. Additionally, many of these tools are open-source and actively incorporate contributions from academic researchers, even when the core development is led by industrial teams.

Major hardware vendors and prominent tech companies have each developed at least one tool to simplify the deployment of ML models on embedded devices. Among the techniques employed in these tools, 8-bit integer quantization is the only one that is widely adopted. The main differentiating factor in performance among these tools is the use of hardware-specific optimizations. Beyond that, the size of the library may also be important, particularly for smaller models.

An ideal tool should give users maximum flexibility, allowing them to bring their own models from popular frameworks, quickly create models from scratch within the tool, or provide data for the tool to automatically discover the best models. Neural Architecture Search (NAS), when designed to account for hardware performance metrics, is a valuable feature for identifying optimal models [95, 96, 97, 98]. In addition to quantization, other optimization techniques, currently absent in most tools, should be considered. These include structured pruning for general use cases, unstructured pruning for accelerators that support it, neuron merging [99], knowledge distillation [100], and early exit networks [101]. The tool should also support a wide range of models and hardware platforms. On-device learning, supported by techniques specifically designed to minimize resource requirements [102, 103, 104], can play a key role in enabling eAI to expand into many new applications. Finally, the tool should be user-friendly, offering a simple and intuitive interface along with detailed documentation and tutorials.

3.7 Automation of eAI Tools

Deploying ML models on embedded devices typically involves several steps, such as model generation, optimization, conversion, and deployment, which we refer to as function blocks. Each step can be time-consuming and rigorous, especially when managing multiple models or devices. This process is further complicated by the limited resources

Table 3.1: Important features of eAI tools.

Tool	Owner	Year	Input	Output	Model	Hardware	Interp based	Free	On-device learning	Ease of use
TFLM	Google	2018	TFLite	C++	DL [93]	kB/MB	✓	✓	✗	⁎
Edge Impulse	Qualcomm	2019	Data, Model	Diverse[1]	ML[2]	kB/MB/GB	✗[2]	✓[3]	✗	⁎⁎⁎
microTVM	Apache	2017	Model	C	DL	kB/MB	✓	✓	✗	⁎⁎
STM32 Cube.AI	ST	2018	Model	C	ML	STM32	-	✓[4]	✗	⁎⁎⁎
Renesas eAI Translator	Renesas	2017	Model	C	DL [49]	Renesas	✗	✓[4]	✗	⁎⁎⁎
Ekkono	Ekkono	2016	Python	C	DL	kB/MB	✓	✗	✓	⁎⁎
ARM-NN	Arm	2018	Model	Android, Linux	DL	MB/GB	✓	✓	✗	⁎⁎
ELL	Microsoft	2017	Model	ELL	DL	Raspberry Pi	-	✓	✗	⁎⁎
NanoEdge AI Studio	ST	2020	Data	C	ML	ARM Cortex-M	✗	✓	✗[5]	⁎⁎⁎

[1] C++ code, library for IDEs, or binary for a specific target.
[2] Edge Impulse can also generate interpreter-based TFLM code.
[3] Some features are only available in upgraded plans.
[4] Free, but not open-source.
[5] It only has on-device learning for anomaly detection.

available on embedded devices, which demand careful consideration of the model's performance metrics and might necessitate multiple iterations of this workflow. To address these challenges, we propose EdgeMark, an open-source automation system that streamlines the deployment of ML models on embedded devices.

EdgeMark simplifies and accelerates the entire workflow of deploying ML models on embedded devices, including the use of eAI tools. It provides an intuitive interface through which users can describe their objective, while automatically managing the underlying complexity. For example, using TFLM often requires users to handle different cases depending on the model's quantization type and input/output dimensions. Additionally, to optimize for memory use, users must register only the operations required by their specific model and configure memory settings manually. EdgeMark abstracts these and other low-level details away, allowing users to focus on model development and performance tuning rather than deployment intricacies.

This system is used, for example, in [85] to quickly generate and evaluate models on a microcontroller, helping to find an efficient model. This work also revealed a potential gap between the estimated performance of models and their actual performance on the target device.

EdgeMark follows a modular design that encompasses the entire pipeline of the aforementioned function blocks. Currently, EdgeMark includes modules whose correspondence to each function block is illustrated in Table 3.2 and Fig. 3.2. These modules are:

- **Generate TF Models:** Generates, compiles, trains, and evaluates TensorFlow models based on user-provided configuration files.
- **Generate Ekkono Models:** As discussed in Section 3.6.6, Ekkono models require users to operate in a specific environment with a different workflow. This module simplifies the process by combining generation, training, evaluation, and conversion into a single step. It uses the same configuration files as the *Generate TF Models* module and primarily produces Ekkono Edge and Ekkono Crystal models.
- **Convert to TFLite:** This module converts TensorFlow models to TFLite format, offering various optimization options during the conversion process.
- **Convert to TFLM:** Converts TFLite models into the TFLM format, generating C++ code for deployment.
- **Convert to Edge Impulse:** This module converts TFLite models into the Edge Impulse format, also resulting in C++ code.
- **Convert to eAI Translator:** Prepares the necessary data files for integration with the eAI Translator's C project.
- **Test on NUCLEO-L4R5ZI:** This module enables testing of TFLM, Edge Impulse, and Ekkono models on the NUCLEO-L4R5ZI board. The respective project will be compiled and flashed to the board. The results will be collected, analyzed, and presented to the user.
- **Test on RenesasRX65N:** This module functions similar to *Test on NUCLEO-L4R5ZI*, but is modified for the Renesas RX65N board.

The modules are interconnected, as shown in Figure 3.2. This modular design offers several advantages. First, it allows users to select only the modules relevant to their specific application, making the system more flexible. For example, if a user is only interested in

generating and converting models to TFLite, they can use the *Generate TF Models* and *Convert to TFLite* modules. It is important to note that the modules must be used in the correct order as illustrated in the figure. Second, it makes the system easily extendable to support new models, tools, or devices. For instance, support for STM32Cube.AI can be added by introducing a new module in the *Conversion* block.

EdgeMark supports TFLM due to its popularity in deploying models on microcontrollers. Similarly, Edge Impulse has been integrated for its extensive features, flexibility, and user-friendly interface. Ekkono is also included in our suite for its lightweight library and unique on-device learning capability, which sets it apart in the field. To ensure a comprehensive comparison of performance, we have included Renesas eAI Translator. This allows us to evaluate general tools like TFLM, which may have a bias toward ARM Cortex-M processors, against more vendor-specific solutions.

With the exception of Ekkono, which uses its own model generation workflow, all other eAI tools work seamlessly with TFLite models. Since TFLite is best supported within the TensorFlow ecosystem, models are primarily developed using TensorFlow and subsequently converted to the TFLite format.

Table 3.2: Correspondence between EdgeMark modules and function blocks.

	Gen	Opt	Conv	Deploy
Generate TF Models	✓			
Generate Ekkono Models	✓		✓	
Convert to TFLite		✓	*	
Convert to TFLM			✓	
Convert to Edge Impulse			✓	
Convert to eAI Translator			✓	
Test on NUCLEO-L4R5ZI				✓
Test on Renesas RX65N				✓

* While this module converts the model into a format suitable for devices like mobile phones, it cannot be directly used with devices supported by EdgeMark.

Running the main script will display the project's main menu where users can select the set of modules they wish to execute. EdgeMark runs the selected modules sequentially, storing their outputs for subsequent modules and for user reference. The primary output from the final layer of the workflow includes the execution time, flash size, RAM usage, and deployment error of the model on the target device. Additionally, the system generates a report summarizing the workflow results.

If generation of models is a part of the selected modules, users need to provide configuration files written in YAML format. These files detail the model architecture, dataset, and specific hyperparameters. Code Snippet 1 is an example configuration file. In this example, each element of `convs_params` is a list of three integers that define the number of channels, kernel size, and stride, respectively. A zero in the number of channels indicates a max-pooling layer, and if the kernel size is also zero, it represents a global average pooling layer. The `denses_params` specifies the number of neurons in each dense layer. The model's output neurons and the loss function will be automatically set based on the dataset. While this configuration scheme supports a broad spectrum of common model architectures, users with more specialized or unsupported designs can still integrate custom

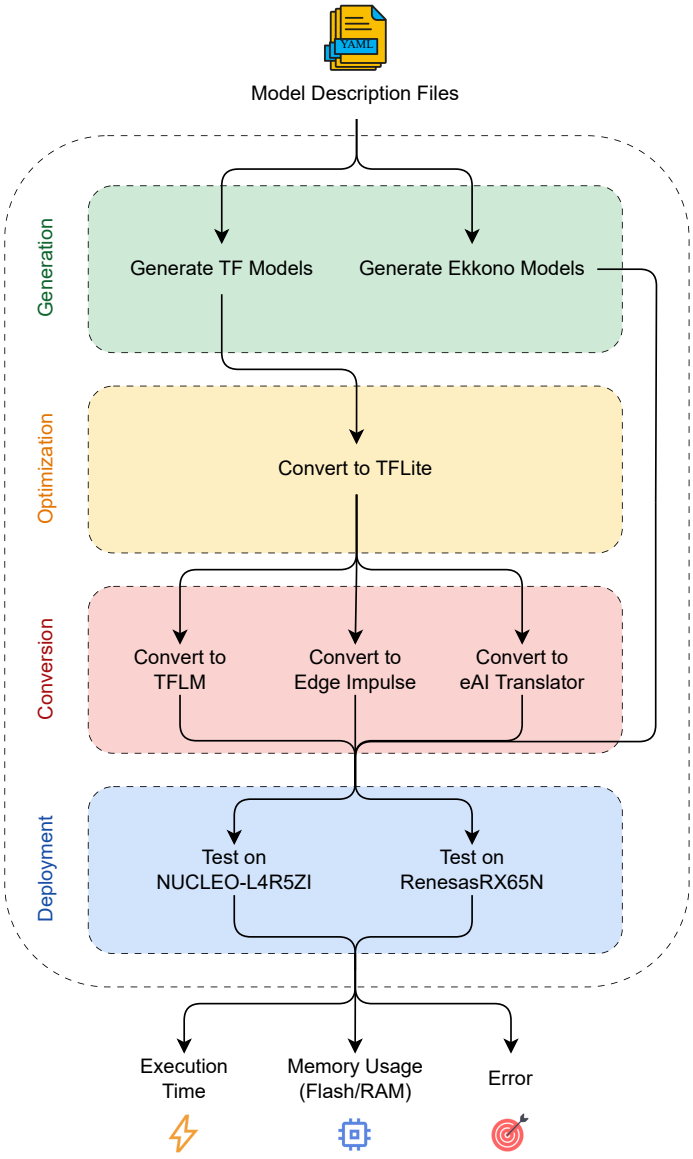


Figure 3.2: EdgeMark modules and their interconnections.

models into EdgeMark thanks to its extensible design.

```yaml
model_type: "CNN"
convs_params: [
    [8, 3, 1],
    [0, 2, 2],
    [16, 3, 1],
    [0, 0, 0]
]
denses_params: [64, 16]
convs_dropout: 0.25
denses_dropout: 0.10
activation: "relu"
use_batch_norm: False
epochs: 50
batch_size: 32
dataset:
    name: "mnist"
    args:
        flat_features: False
random_seed: 42
```

Code Snippet 1: Example of a user-defined configuration file for generating a CNN model.

EdgeMark currently supports various models, such as FC, Convolutional Neural Network (CNN), and Recurrent Neural Network (RNN), along with specialized models like MLPerf Tiny models. The system includes datasets like MNIST, CIFAR-10, ImageNet-V2, Visual Wake Words, Boston Housing, and more. In addition to TFLite conversion, users can choose among TFLM, Edge Impulse, Ekkono, and Renesas eAI Translator for converting models to source code. The models can be tested on NUCLEO-L4R5ZI and Renesas RX65N boards. The complete list of supported models, datasets, tools, and hardware is presented in Table 3.3. It is important to note that, EdgeMark is designed for easy extensibility, allowing new models, datasets, tools, and devices to be incorporated with minimal effort.

Adding a new model or dataset to EdgeMark is straightforward. It involves subclassing the appropriate superclass and implementing a few required methods. These superclasses are well-documented and designed to minimize constraints on developers. To add new eAI tools to EdgeMark, custom integration methods may be required, as each tool might function differently. However, most tools accept a TensorFlow or TFLite model–produced by previous modules–as input and generate C/C++ code as output, which keeps their integration relatively consistent. To fully automate the process for each tool and hardware combination, a corresponding C/C++ project must be created. For this purpose, EdgeMark provides template files that are also well-documented and users need to complete them. In practice, new models, datasets, tools, or hardware often behave similarly to existing ones, so previously implemented components usually serve as helpful references or starting points.

In the remainder of this section, we delve deeper into the functionality of each module. *Generate TF Models* generates a TensorFlow model for each configuration file available in a specific directory. This process results in the storage of TensorFlow models, their corresponding best and last weights, and training and evaluation logs, which can be reviewed using TensorBoard and Weights & Biases (W&B), along with some supplementary files. Notably, in this module, users can select random dataset generators to create datasets

Table 3.3: EdgeMark support range for models, datasets, tools, and hardware.

Models	FC	
	CNN	
	RNN	SimpleRNN
		LSTM
		GRU
	MLPerf Tiny	AE
		DS CNN
		ResNet
		MBNet
	MBNetV2	
Datasets	MNIST	
	CIFAR-10	
	ImageNet-V2	
	Boston Housing	
	Visual Wake Words	
	Shakespeare	
	Sinus	
	ToyADMOS	
	RandomSet	Classification
		Regression
		Sequence
Tools	TFLM	
	Edge Impulse	
	Ekkono	
	Renesas eAI Translator	
Hardware	STM	NUCLEO-L4R5ZI
	Renesas	RX65N

with their desired characteristics. These generators allow users to complete the system's workflow and enable evaluation of performance metrics like execution time, memory usage, and deployment error, regardless of model correctness or accuracy.

The *Convert to TFLite* module is responsible for transforming TensorFlow models into TFLite models. Beyond standard conversion, this module provides a range of optimization options, including six types of quantization (such as 8-bit integer and dynamic range quantization), pruning, clustering, and a collaborative optimization strategy that combines multiple optimizations.

Convert to TFLM/Edge Impulse/eAI Translator converts TFLite models into their platform's specific C/C++ code. To minimize memory consumption, TFLM requires users to include only the operators necessary for running the model. The *Convert to TFLM* module automates this process by identifying the operators used in the model and including them in the generated code.

Minimizing the memory footprint of the generated code is especially critical for embedded deployments. While it is sometimes possible to run non-optimized models on target devices, memory optimization is often essential in production settings. Reducing memory usage not only enables the coexistence of other system components but also provides room to deploy more complex models, potentially improving overall system performance and capability.

TFLM employs a static memory allocation strategy, which means users must define an arena size–a fixed memory buffer where all allocations occur during model execution. Determining the minimum viable arena size involves two steps: first, estimating the memory usage of the model, and second, running a search algorithm to refine the value. The goal is to find the smallest arena size that allows the model to run without encountering memory errors.

The initial memory estimation is based on the maximum memory demand of consecutive model layers. Specifically, it considers the sum of the output sizes of two consecutive layers that must coexist in memory, assuming no branching in the model architecture. This estimation provides a starting point for the search process.

To accurately determine the minimum arena size, we implement a search algorithm, a simplified version of which is outlined in Algorithm 1. The algorithm refines the arena size in increments determined by a resolution of 0.5, 1, or 2 kB, depending on the estimated value. While the initial estimate is expected to be close to the actual minimum in most cases, significant deviations can occur in certain scenarios. During experiments, the growth and shrinking parameters of the algorithm can be fine-tuned to accelerate convergence and improve efficiency in finding the optimal arena size.

For convenience, the *Generate Ekkono Models* module accepts the same configuration files as the *Generate TF Models* module. However, it is important to note that Ekkono supports only a limited subset of TensorFlow models. Specifically, it supports FC models for regression tasks, excluding features like dropout or batch normalization. Additionally, Ekkono does not support optimization techniques such as quantization, pruning, or clustering. Once the models are generated, they are trained and evaluated using the Ekkono-specific workflow, with the resulting Ekkono Edge and Ekkono Crystal models stored as output.

To deploy these models on target devices, generic C/C++ projects are prepared for each platform and hardware. The generated code for models in previous modules adheres to

Algorithm 1 Find Minimum Arena Size

```
growth_rate ← 1.25
shrink_rate ← 0.8
growth_steps ← 4
shrink_steps ← 4
Initialize lower_bound, upper_bound, best_guess ← None

function UpdateBounds(guess)
    try guess on embedded device
    if guess is too low then
        lower_bound ← guess
    else if guess is too high then
        upper_bound ← guess
    else if guess is correct then
        best_guess ← guess
        upper_bound ← guess
    end if
end function

UpdateBounds(estimated_arena_size)

while upper_bound - lower_bound ≠ 1 do
    if lower_bound exists and not upper_bound then
        next_guess ← max(lower_bound × growth_rate, lower_bound + growth_steps)
    else if upper_bound exists and not lower_bound then
        next_guess ← min(upper_bound × shrink_rate, upper_bound − shrink_steps)
    else
        next_guess ← (lower_bound + upper_bound) / 2
    end if

    UpdateBounds(next_guess)
end while
```

the structure of these general projects; for example, they define the model's input and output size and type. Thus, for *Test on NUCLEO-L4R5ZI* and *Test on RenesasRX65N*, changing the model is as simple as replacing the model's source code with the new one. Afterwards, they compile and flash the project to the board, where the model is executed, and the results are collected. These results are then analyzed and presented to the user in a report, which includes the following key metrics:

- **Execution time:** This metric measures how long it takes for the model to run on the target device. It is measured by iteratively running the model 10 times and taking the average. The same input was used across all iterations. In our testing environment, changing the input, having a warm-up phase, or re-running the test have negligible to no effect on the results. The standard deviation of the execution time was found to be close to zero, reflecting the consistency of the measurement.
- **Flash size:** This represents the amount of flash memory required by the model. The total flash memory usage is reported by the compiler, and the flash size for the model and its dependencies is obtained by subtracting the flash memory usage of a base project (a similar project without the model and its dependencies) from that of the project including the model.
- **RAM usage:** Similar to flash size, RAM usage quantifies the memory consumed by the model during execution. It is also based on the compiler's report, since all tools supported by EdgeMark follow a static memory allocation strategy.
- **Deployment error:** This is the error rate of the model when deployed on the target device. The deployment error is computed as the average normalized absolute difference between the model's output on the target device and its expected output generated by running the model on a PC. The model on PC can be either the basic version or optimized version (e.g., quantized version). We assume optimization as a part of conversion and always use the basic version of the model on PC. To calculate deployment error, we provide the model with 10 different inputs and report the average error rate.

3.8 Benchmark of eAI Tools

To evaluate the performance of the eAI tools, we conducted a series of experiments using EdgeMark. Additionally, we analyzed the impact of various factors, such as different quantization schemes and the presence of a floating-point unit (FPU). In total, we have performed thousands of tests, with each test taking an average of two to three minutes to complete, the majority of which was spent on cross-compilation[1] and flashing the project to the boards.

EdgeMark employs several tools and platforms to accomplish its objective. Table 3.4 summarizes their versions and purposes as used in the benchmark. Among these, STM32CubeIDE, STM32CubeCLT, Renesas e2 studio, and Renesas Flash Programmer, must be installed manually if the relevant modules are used. Additionally, an Edge Impulse account is necessary to utilize its corresponding module.

For both IDEs, STM32CubeIDE and Renesas e2 studio, the projects were created using default settings, with the exception of the compiler optimizations, which were set to their maximum level. To execute TFLM models on the NUCLEO-L4R5ZI board, we integrated the CMSIS-NN library, which significantly accelerates the performance of 8-bit and 16-bit integer quantized models by leveraging the SIMD instructions of the ARM Cortex-M4

[1] The cross-compilation was done on a machine with an Intel Core i7-11850H CPU.

Table 3.4: Tools and Platforms Used in EdgeMark

Name	Version	Purpose
TensorFlow	2.15.0	Model generation
TFLite	Coupled with TensorFlow	Model optimization and conversion
TFLM	Commit 42f4bb8	Model conversion
Edge Impulse	1.56.13 date: 12-09-2024	Model conversion
Ekkono	23.10	Model generation and conversion
Renesas eAI Translator	3.2.0	Model conversion
STM32CubeIDE	1.14.1	Project setup and compilation
STM32CubeCLT	2.16.0	Flashing projects to the board
Renesas e2 studio	24.1.1	Project setup and compilation
Renesas Flash Programmer	3.15.00	Flashing projects to the board

processor. Renesas has also provided a CMSIS-NN library optimized for their RX microcontrollers, which we have used to execute eAI Translator models on the Renesas RX65N board.

3.8.1 Models

We selected a wide range of models to ensure a comprehensive evaluation of the tools' capabilities. These include 11 FC models, 7 CNN models, 7 RNN models, and 4 models taken from the MLPerf Tiny benchmark. As illustrated in Fig. 3.3, the FC and CNN models are well distributed across a range of parameters and Multiply-Accumulates (MACs), ensuring that our evaluation covers both lightweight and computationally intensive models.

FC 0 serves as a minimal baseline, with an input size of 1, an output size of 1, and no hidden layers. It can be useful for analyzing the overhead of the library executing the model. Similarly, in the RNN category, *Simple 0*, which has the minimal structure of an RNN, plays an analogous role. However, because RNN models involve more complex operations, *Simple 0* is much more demanding compared to *FC 0*.

The RNN models include five SimpleRNNs, one LSTM, and one GRU. All RNN models process sequence lengths of 100, except for *Simple 0*, which operates on a sequence length of 2. Two of the SimpleRNN-based models are *Shakespeare 1* and *Shakespeare 2* models that have an embedding layer before their RNN units and are trained on the `tiny_shakespeare` dataset [105]. Detailed information about the RNN models is presented in Table 3.5.

Lastly, Table 3.6 provides an overview of the MLPerf Tiny models used in this evaluation. By including this wide range of models, we aim to provide meaningful insights into the performance and efficiency of the tools across different neural network architectures.

3.8.2 Experiments

We have structured our analysis into ten experiments, five of which are described in this section, with the remaining five detailed in 3.10. It is worth emphasizing that each experiment is designed to compare the general behavior of the test subjects, though there may be some exceptional cases that deviate from the trends. For a comprehensive overview

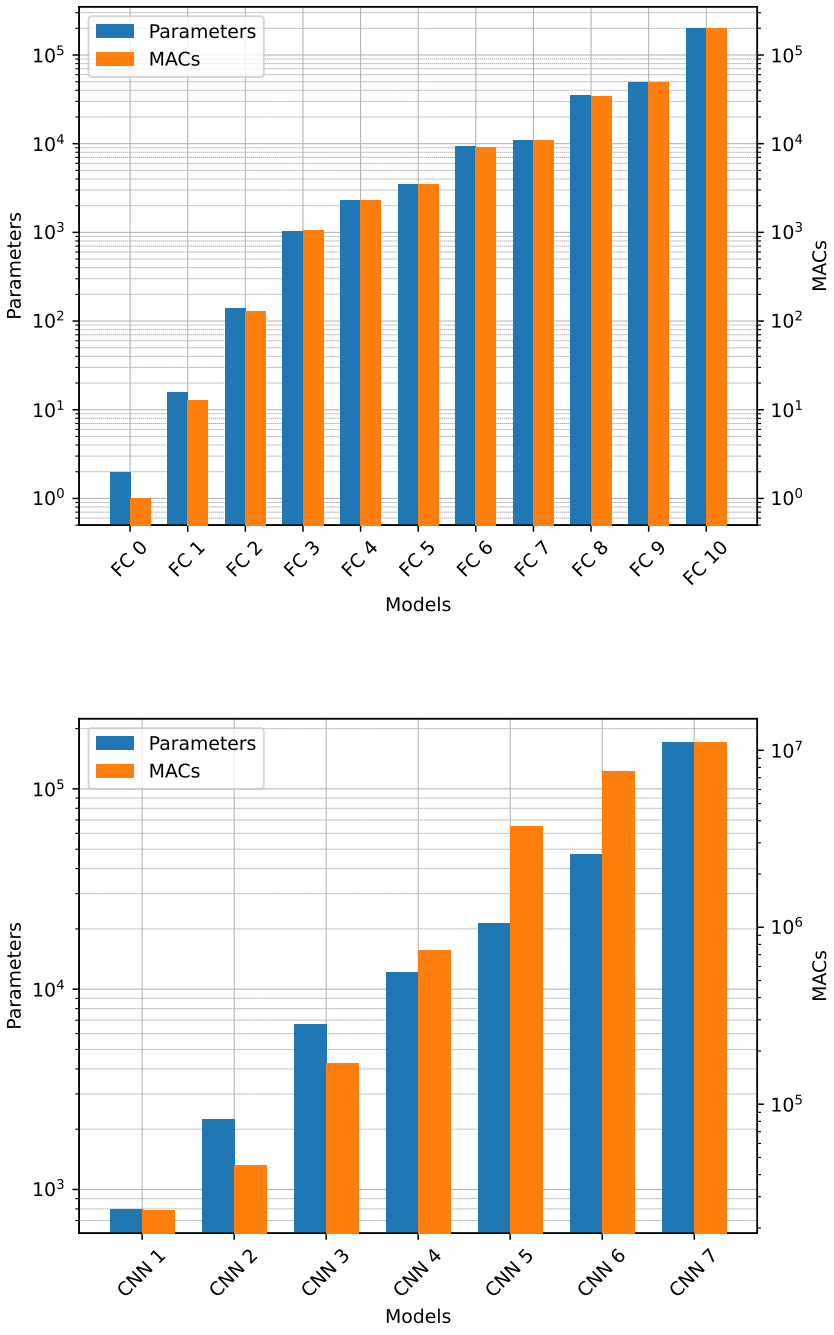


Figure 3.3: Number of parameters and MACs for FC and CNN models.

Table 3.5: Details of RNN Models

Model	RNN Units	RNN Input	Params	MACs
Simple 0	1	1	5	9
Simple 1	64	32	8,288	827,200
Simple 2	128	64	32,960	3,292,800
Shakespeare 1	64	32	12,513	1,056,300
Shakespeare 2	128	64	37,249	3,321,900
LSTM	64	32	26,912	2,702,400
GRU	64	32	20,896	2,094,400

Table 3.6: Details of MLPerf Tiny Models

Model	Params	MACs	Application
AE (Autoencoder)	269,992	269,156	Anomaly detection
DS CNN (Depthwise Separable CNN)	24,908	2,696,804	Keyword spotting
ResNet	78,666	12,561,054	Image classification
MobileNet	221,794	7,606,598	Image classification

of the results, readers are encouraged to refer to the project's documentation[2].

In general, each experiment focuses on the following discussion points: *Model correctness*, which examines whether any models failed to execute or produced unacceptable deployment errors; *execution time*, *flash size*, and *RAM usage*, which provide comparative insights into the respective metrics across study elements; and, finally, a *conclusion* that summarizes the overall performance and behavior of the subjects under consideration.

Quantizations

In general, the following quantization schemes are available in EdgeMark:

- **Dynamic:** Dynamic range quantization. Weights are quantized to 8-bit integers, while activations are stored in 32-bit floats. To accelerate inference, activations can be dynamically quantized to 8-bit integers and then dequantized back to 32-bit floats for storage [106].
- **Int8:** Full integer quantization. Both weights and activations are quantized to 8-bit integers. However, the input and output remain in 32-bit floats [107].
- **Int8 only:** Similar to *int8*, but in this case, everything is quantized to integers without fallback to floats [107].
- **16x8:** To improve the accuracy of the quantized model, the activations are quantized to 16-bit integers while the weights are quantized to 8-bit integers. Similar to *int8*, the input and output are left in 32-bit floats [108].
- **16x8 int only:** Same as *16x8*, but everything is quantized to integers without fallback to floats [108].
- **Float 16:** Weights are quantized to 16-bit floats, reducing model size with minimal impact on accuracy [109].

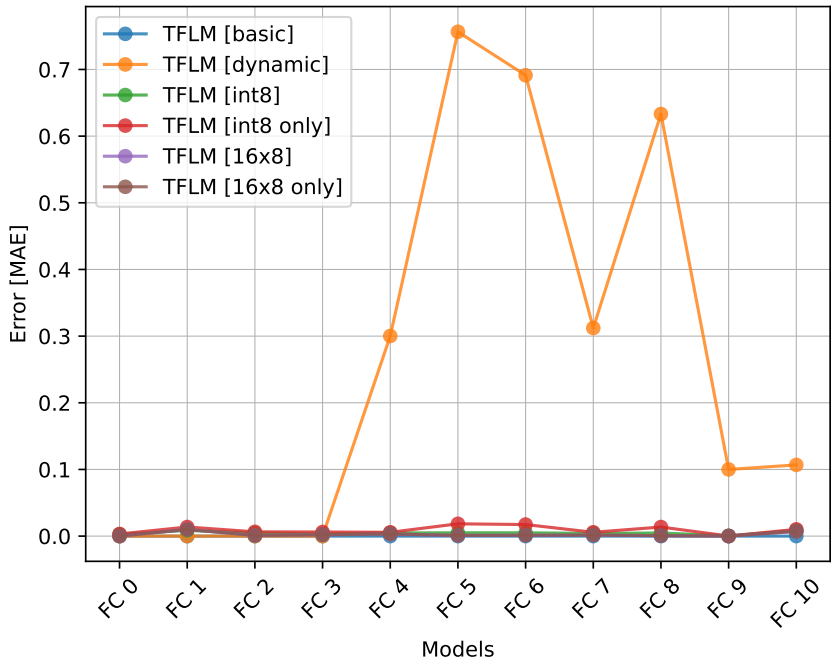
[2] https://black3rror.github.io/EdgeMark

In this study, we evaluate the performance of several quantization schemes using the TFLM platform. TFLM was chosen because it is the only framework that supports the majority of available quantization methods. In particular, it supports all the abovementioned quantizations, except *float 16*. This is while, Edge Impulse and eAI Translator only support the *int8 only* quantization[3].

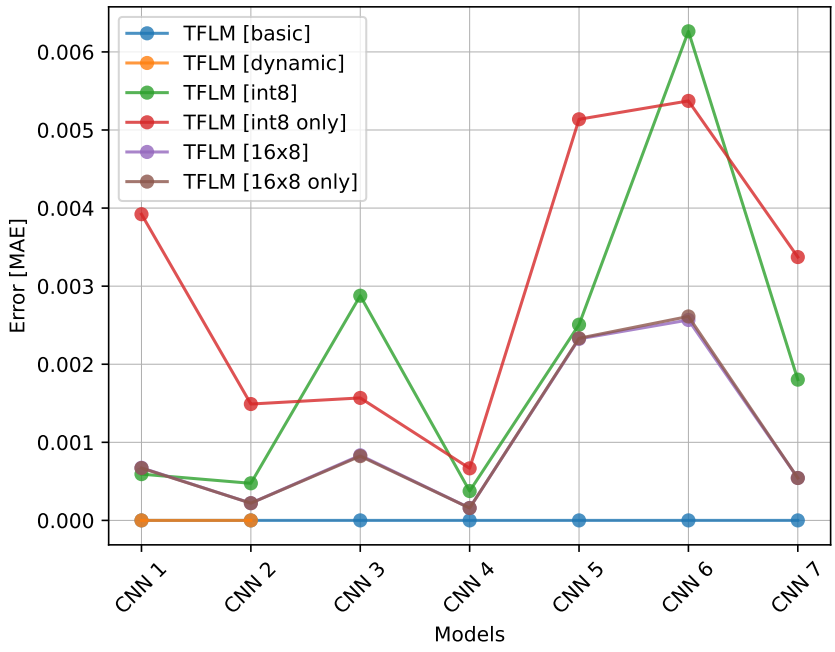
To perform the evaluation, we tested the FC and CNN models on the two available boards. Figures 3.4–3.7 compare these setups in terms of deployment error, execution time, flash size, and RAM usage on the STM board.

- **Model correctness:**
  - We recommend avoiding using *dynamic* quantization since it lacks proper support (evidenced by missing data points and high error rates in Fig. 3.4). Models using this scheme often fail to run or exhibit unacceptably high errors. Even when deployment succeeds, it does not outperform other quantization schemes in any metric.
  - During this and some other experiments, some models could not be executed successfully on the Renesas board, with the program unexpectedly stopping during execution. The root cause of this behavior remains unclear.
  - Fig. 3.4 shows that all other quantization schemes reach an acceptable error. The *basic* conversion perfectly produces the expected outputs, while other types of quantization introduce a bit of error rooted in the nature of quantization. Between *int8* and *16x8* quantization schemes, as expected, *16x8* shows smaller error rates due to its higher precision representation of activations.
- **Execution time:** Although, based on the results illustrated in Fig. 3.5, determining the fastest quantization scheme may not be clear, the following statements appear to summarize the key points. For FC models, the *int8 only* and *16x8 int only* quantizations outperform their non-only counterparts, with *16x8 int only* performing best for smaller models and *int8 only* excelling for larger models. For CNN models, *basic* and *dynamic* quantizations are significantly slower due to their lack of CMSIS-NN library optimizations. The *int8* and *16x8* variants have similar performance, with *16x8 int only* slightly outperforming on smaller networks and *int8 only* proving better for larger ones.
- **Flash size:** As shown in Fig. 3.6, all quantization schemes start with relatively similar flash sizes, but as the model scales, the *basic* version's flash size increases fourfold (four bytes per parameter), whereas the other variants increase by only one byte per parameter. Therefore, the *basic* model gets worse as model size increases.
- **RAM usage:** Based on Fig. 3.7, for CNN models, as expected, the *int8*-based variants require less RAM than other types of the same model, since they store activations in 8-bit representation. These are followed by *16x8*-based models, and finally, *basic* and *dynamic* schemes requiring the most RAM. For FC models, RAM consumption patterns are less predictable, though the *int8 only* scheme consistently proves efficient, often matching or outperforming other methods.
- **Conclusion:** The choice of quantization scheme depends on many factors and can lead to execution that is a couple of times faster and requires less memory. Particularly for CNN models, if RAM is a constraint, int8 variants are preferable.

[3]The *int8* quantization may also be supported, but it was not tested.

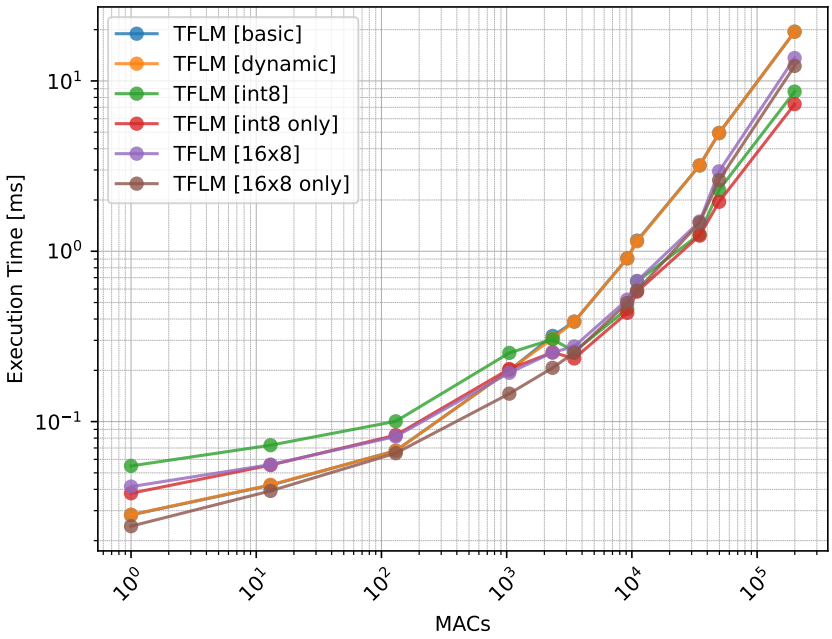


(a) FC models.

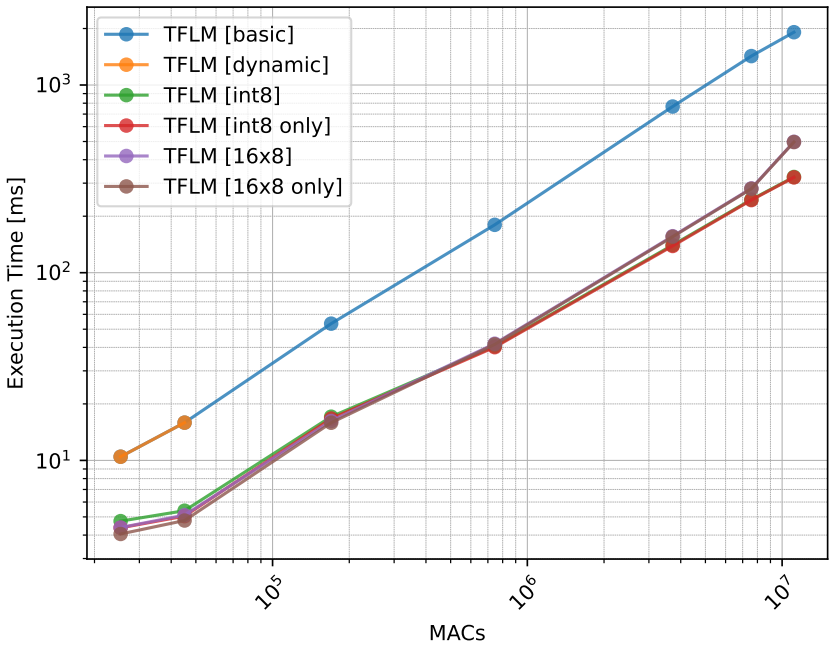


(b) CNN models.

Figure 3.4: Deployment error across various quantization schemes. The models were tested on NUCLEO-L4R5ZI.

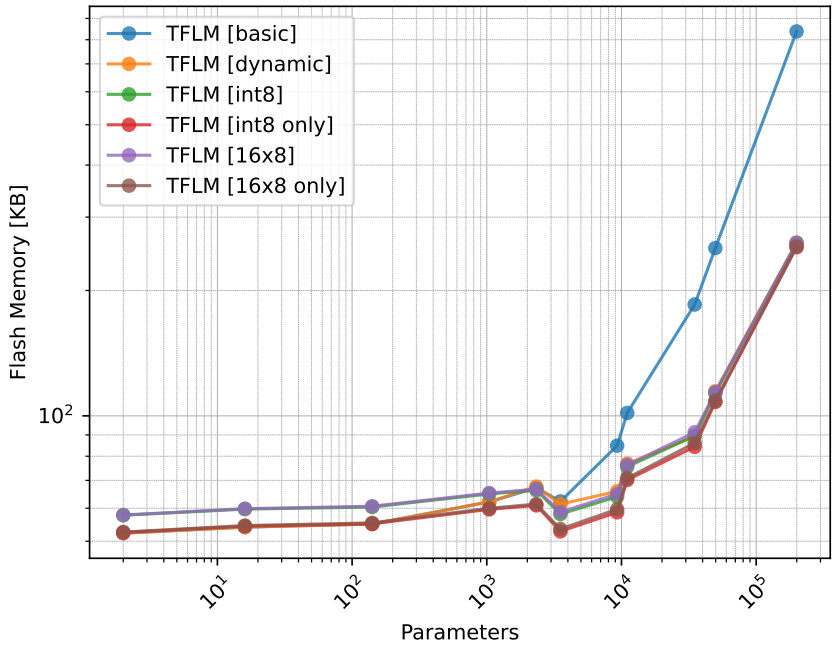


(a) FC models.

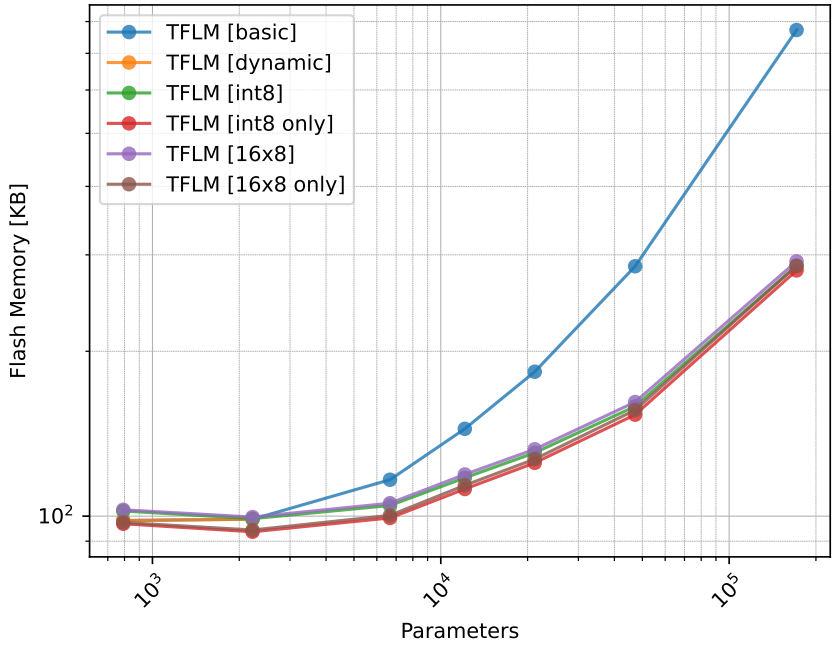


(b) CNN models.

Figure 3.5: Average execution time across various quantization schemes. The models were tested on NUCLEO-L4R5ZI.



(a) FC models.



(b) CNN models.

Figure 3.6: Flash memory usage for various quantization schemes. The models were tested on NUCLEO-L4R5ZI.

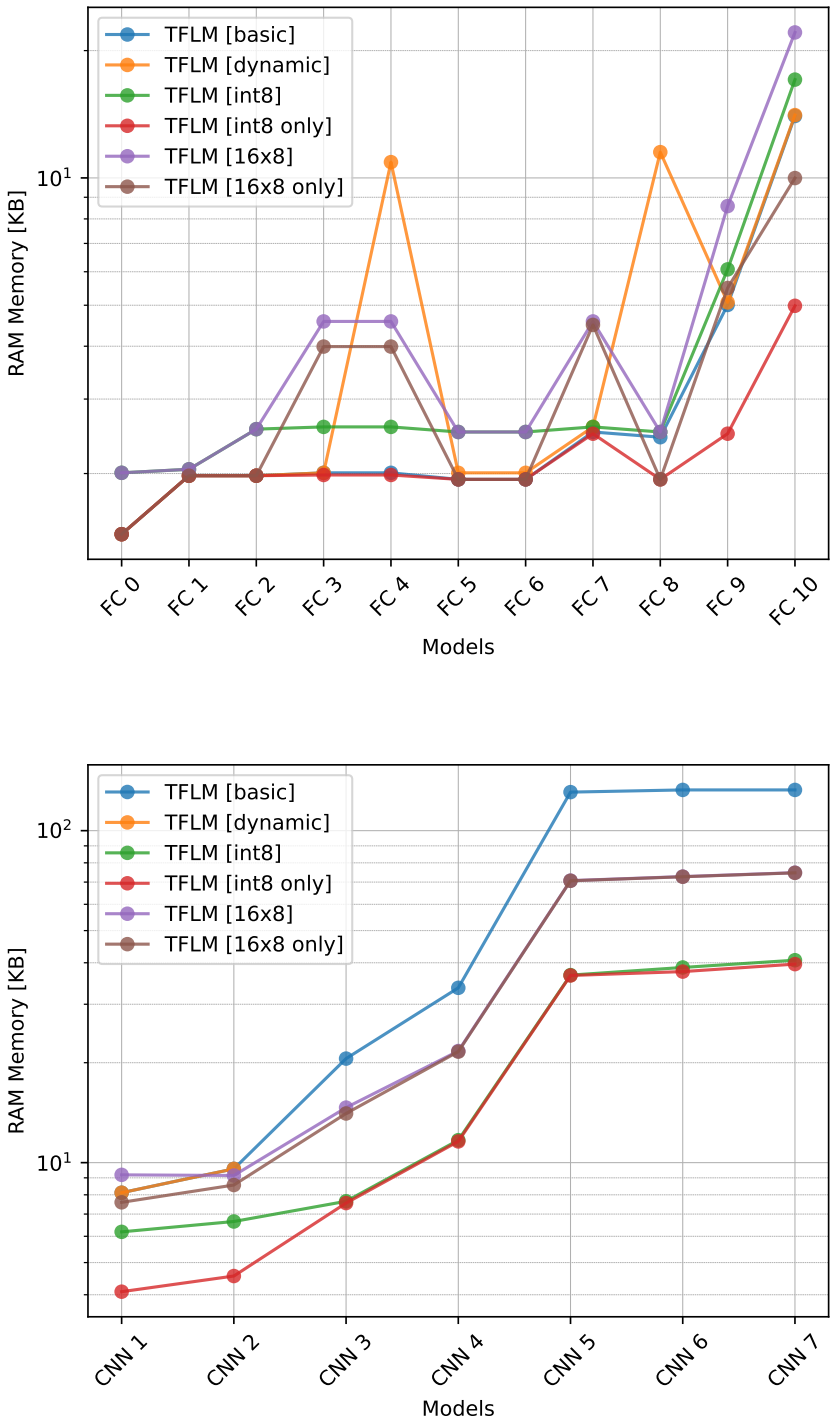


Figure 3.7: RAM usage across various quantization schemes. The models were tested on NUCLEO-L4R5ZI.

Generally, *int8 only* quantization is a good choice for most cases. Together with the *basic* (non-quantized) variant, these two schemes will serve as the default choices for the remainder of this study.

Pruning and Clustering

This study examines the effects of pruning and clustering on model performance. After quantization, pruning is the most popular technique for reducing the size of a model. However, only a few sources make it clear that unstructured pruning will not bring any benefit to general-purpose processors. To leverage this technique, either specialized hardware is required, or pruning must be structured–eliminating entire units of the model, such as neurons or channels, thereby creating a model with a modified architecture.

The counterintuitive nature of this limitation lies in the fact that, in unstructured pruning, the zero values for the pruned weights still occupy memory and consume the same storage space as non-pruned weights. It is worth noting that pruned models can be compressed more effectively due to the repetitive zero values in their weights. However, such compression requires decompression before execution, which negates most practical gains. Moreover, in terms of runtime performance, multiplying a pruned weight (zero) with another value takes the same clock cycles as a multiplication involving non-zero weights. Consequently, unstructured pruning yields no improvement in execution time. Similarly, clustering, which involves grouping weights into discrete clusters, offers little to no advantage in improving the model's performance on general-purpose hardware.

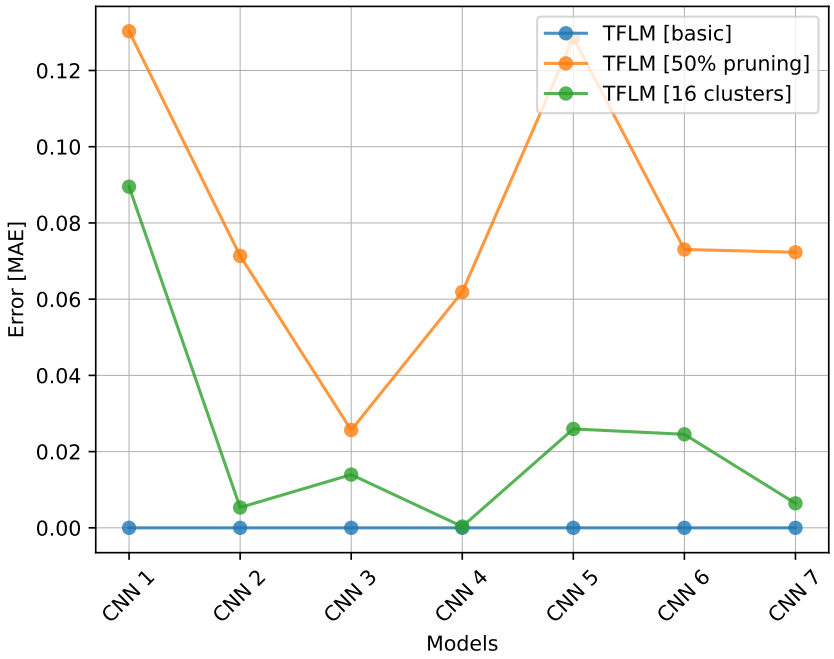
To elucidate these findings, we evaluated FC and CNN models on the NUCLEO-L4R5ZI board. In our experiments, we applied 50% sparsity for pruning and clustering with 16 centroids. These techniques increase the error of the models (see Fig. 3.8a), which can be mitigated to some extent through fine-tuning. However, as illustrated in Fig.3.8b, the execution times for the basic, pruned, and clustered models are nearly identical, with less than 0.01% difference. This negligible variation means the plots are essentially overlaid. The same holds true for the flash memory usage and RAM requirements of the models.

TFLM vs. Edge Impulse

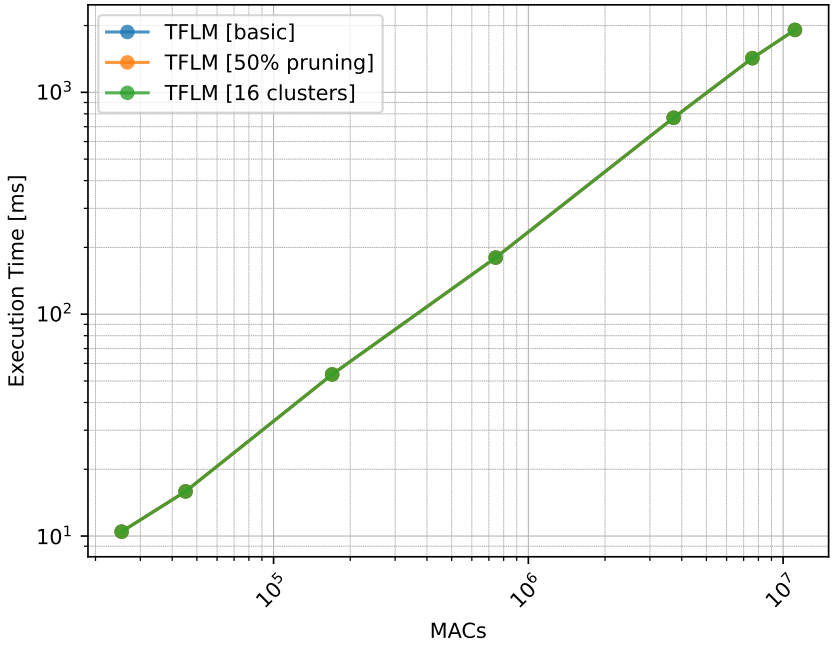
TFLM and Edge Impulse are two prominent tools in the eAI domain. While Edge Impulse originally relied on TFLM and still provides the option to deploy models using it, it has significantly evolved over the years. With features like its EON compiler, Edge Impulse introduces a unique approach to model execution. However, it is worth noting that Edge Impulse still utilizes TFLM's core kernels in most cases.

As Edge Impulse does not natively support RNNs, this study focuses on testing FC, CNN, and MLPerf Tiny models deployed on the STM and Renesas boards. The results show that on the Renesas board, TFLM outperforms Edge Impulse in terms of execution time, flash size, and RAM usage. On the STM board, the key findings are as follows:

- **Model correctness:** Since the MBNet model from the MLPerf Tiny test suite was too large in most test cases, it has been excluded from the comparison. Quantized models introduced a minor degree of error in the outputs, though this error was negligible and within acceptable margins. Notably, Edge Impulse's models were more accurate in producing the expected outputs compared to TFLM.
- **Execution time:** TFLM demonstrated faster execution times for smaller FC models. For other models, the two were in par, with a slight edge for TFLM.
- **Flash size:** Edge Impulse consumed less flash memory compared to TFLM. The extent of this difference varied based on the model's type and size, ranging from negligible to over 100% improvement. The differentiating factor is the library's base

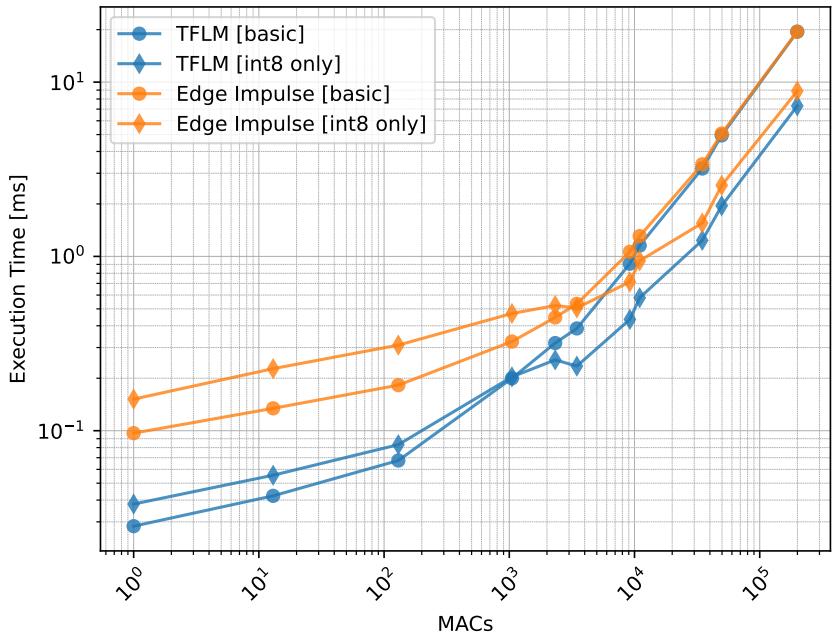


(a) Deployment error.

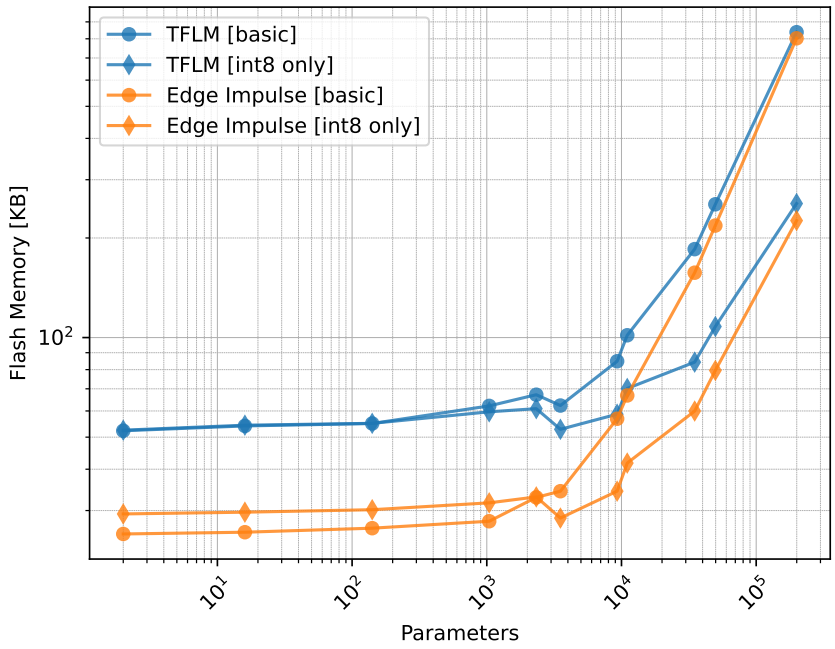


(b) Execution time. The three plots are drawn on top of each other, appearing as one.

Figure 3.8: Performance evaluation of basic, pruned, and clustered CNN models on the NUCLEO-L4R5ZI board.



(a) Execution time.



(b) Flash size.

Figure 3.9: Performance evaluation of FC models deployed using Edge Impulse and TFLM on the STM board.

size, and the amount of improvement diminishes as the model grows larger.

- **RAM usage:** For small to medium-sized FC models, Edge Impulse showed better RAM efficiency. For other model types, the memory usage of both libraries was nearly equivalent.
- **Conclusion:** On the Renesas RX65N board, TFLM is the preferred tool for deployment. On the NUCLEO-L4R5ZI board, TFLM is recommended when execution speed is a priority, while Edge Impulse may be more suitable in applications where flash size or RAM usage is more critical.

TFLM vs. Ekkono

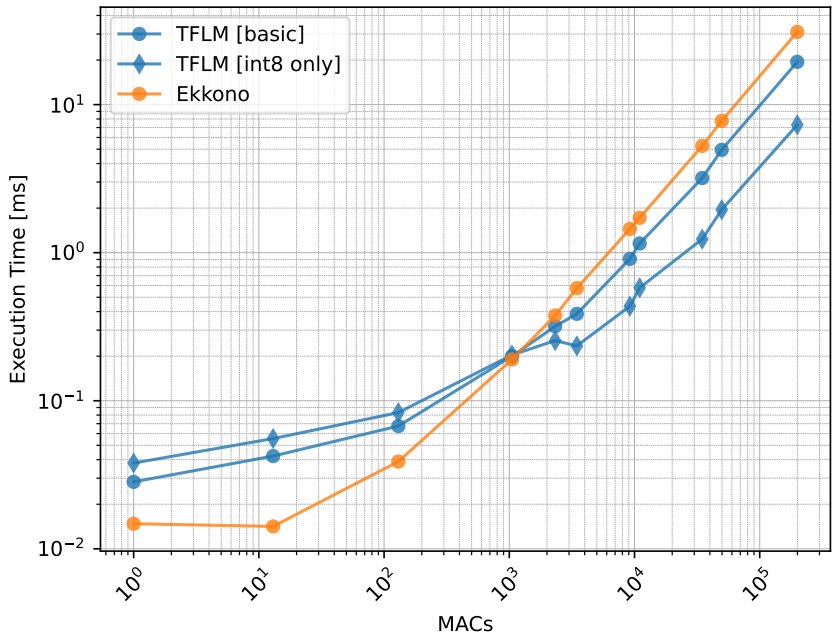
Ekkono is a lightweight edge AI library designed to support incremental learning directly on embedded devices. Since Ekkono is not able to do classification, we have slightly changed some models to make them suitable for regression. The changes are minimal and should not have a noticeable impact on the results.

- **Model correctness:** Both Ekkono and TFLM's *basic* version can perfectly reproduce the expected values. TFLM's *int8 only* version has a bit of error which remains in the acceptable range.
- **Execution time:** As shown in Fig. 3.10a, Ekkono delivered faster execution times for smaller models. In contrast, TFLM performed better with larger models, particularly when using the *int8 only* version.
- **Flash size:** As illustrated in Fig. 3.10b, Ekkono, being a simpler and more lightweight library, offers a smaller baseline flash memory footprint than TFLM (18.8 kB vs. 52.2 kB). However, as model size increases, the performance gap narrows, with the *basic* version of TFLM catching up. This is while the *int8 only* version of TFLM surpases their performance and becomes the best choice for large models.
- **RAM usage:** Similar to flash size trends, Ekkono is more efficient with smaller models, while TFLM's *int8 only* variant performs better as models grow in size. Nevertheless, the memory requirements for all tested FC models were relatively modest (maximum 14 kB) and are unlikely to constrain most applications.
- **Conclusion:** For smaller models, Ekkono provides a more efficient solution. However, as model size increases, TFLM's *int8 only* variant emerges as the more effective choice.

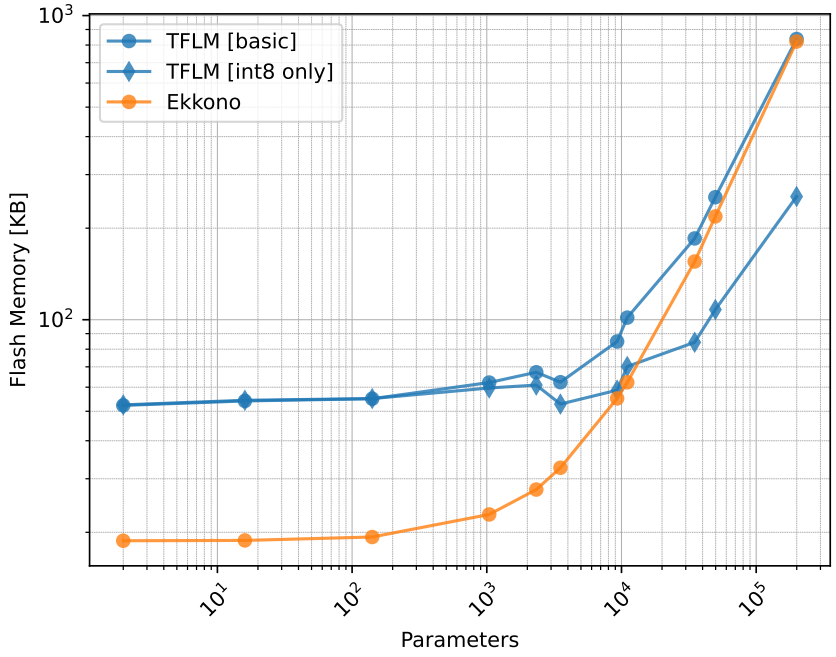
TFLM vs. eAI Translator

Renesas eAI Translator is a vendor-specific tool designed to enable the execution of neural networks on Renesas microcontrollers. Our testing hardware, the Renesas RX65N, incorporates a proprietary RXv2 CPU architecture. Since TFLM lacks optimized kernel implementations for RXv2, it handles this architecture as a general-purpose CPU. In contrast, Renesas eAI Translator provides the capability to fine-tune implementations specifically for the target hardware, potentially achieving better performance. In this study, we assess whether such hardware-specific tuning allows eAI Translator to outperform the more generic TFLM.

- **Model correctness:** The error rate of TFLM and eAI Translator are normally the same, but for some large models, TFLM is more suitable due to lower error rates.
- **Execution time:** Renesas has optimized the CMSIS-NN library to align with the hardware specifications of RX microcontrollers. We utilized this optimized library with eAI Translator but not with TFLM. As a result, we exclude the *int8 only* variants

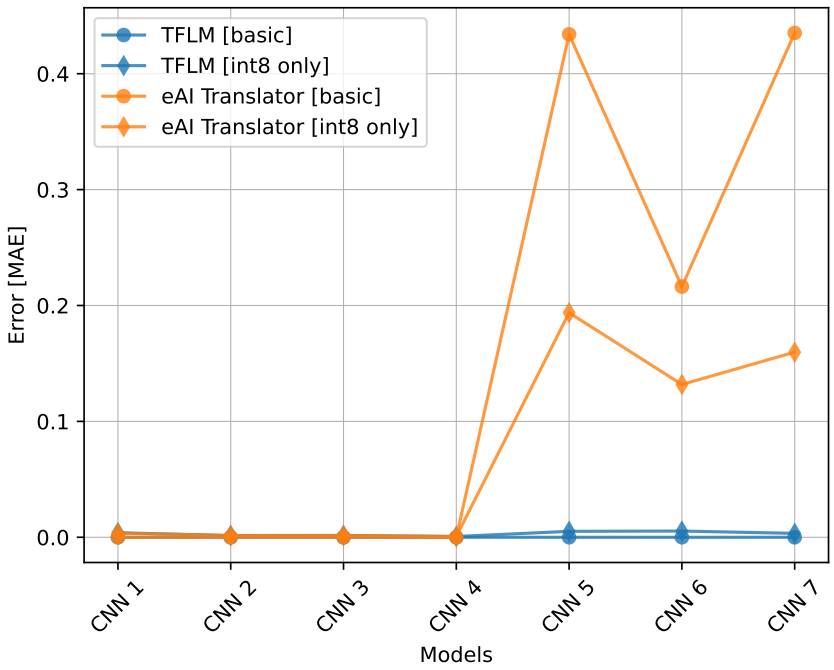


(a) Execution time.

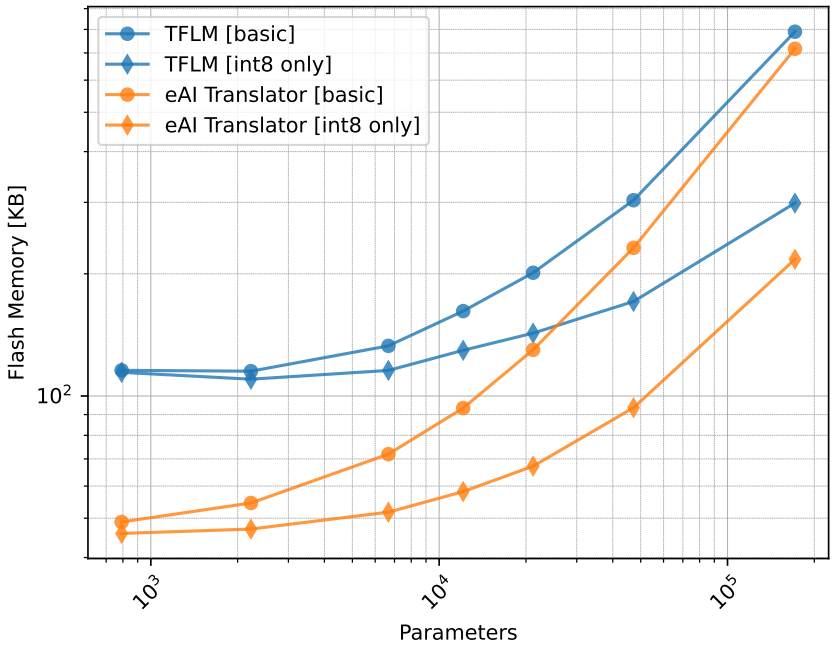


(b) Flash size.

Figure 3.10: Performance evaluation of Ekkono and TFLM models on the NUCLEO-L4R5ZI board.



(a) Deployment error.



(b) Flash size.

Figure 3.11: Performance evaluation of CNN models deployed using eAI Translator and TFLM on the Renesas RX65N board.

of both platforms from the execution time comparison. Focusing on their *basic* variants, our results show that eAI Translator generally outperforms TFLM, particularly on smaller models.

- **Flash size:** The eAI Translator library requires significantly less flash memory compared to TFLM. This makes eAI Translator a stronger candidate for use in memory-constrained environments, especially when dealing with smaller models.
- **RAM usage:** In terms of RAM consumption, eAI Translator demonstrates slightly better efficiency than TFLM.
- **Conclusion:** Overall, except in a few cases highlighted above, eAI Translator is often the better choice across all evaluated metrics, including execution time, memory usage, and flash size.

3.9 Conclusion

This paper explores the landscape of eAI tools, introduces an automation system to streamline their usage, and benchmarks their performance across various metrics. The review identifies key features of these tools, revealing that while they generally support techniques like quantization, many advanced methods in the field remain absent.

To address the challenges in evaluating eAI tools, we developed EdgeMark, an automation system designed to simplify model deployment workflows. EdgeMark provides a unified interface for deploying models on target devices, automating the entire process of model generation, optimization, conversion, and deployment. The system is user-friendly, modular, and easily extensible, enabling researchers and developers to test their ideas with minimal effort.

The benchmarking results showed that the performance of eAI tools varies across different metrics. The choice of quantization scheme, for example, can significantly impact execution time, flash size, and RAM usage. The study also compared the performance of TFLM with Edge Impulse, Ekkono, and Renesas eAI Translator. The results showed that the choice of tool depends on the application requirements, with each tool having its own strengths and weaknesses.

The insights from this work can guide developers in selecting the most suitable eAI tools based on their specific application requirements, while the automation system lowers the adoption barrier for such technologies. Future work will focus on expanding the benchmark to include more tools and platforms, as well as incorporating energy measurements to provide a more comprehensive evaluation of eAI tools. An interesting area for further exploration is evaluating these tools on hardware accelerators specifically designed for neural network processing. Further, the automation system can be used in NAS research to provide real measurements of the performance of generated models.

As embedded systems increasingly rely on AI-driven capabilities, this study provides a foundational step toward creating streamlined workflows and taking informed decisions in selecting and deploying eAI tools, ultimately advancing the integration of AI into resource-constrained devices.

Acknowledgements

This work was supported by the Innovation Fund Denmark for the Project Digital Research Centre Denmark (DIREC) under Grant 9142-00001B.

3.10 Additional Experiments

3.10.1 RNNs

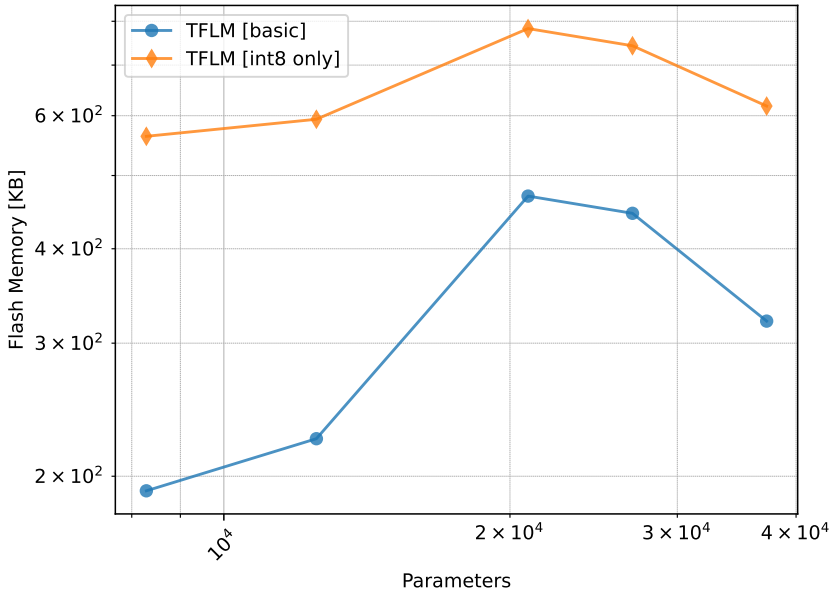
RNN models are widely used in applications that require processing sequential data. However, deploying these resource-intensive models on embedded devices requires careful consideration. In this study, we evaluated the performance of SimpleRNN, LSTM, and GRU models using TFLM on the NUCLEO-L4R5ZI board. The models are introduced in Section 3.8.1, and the results can be found in Table 3.7. To provide a clear comparison, memory usage is visualized in Fig.3.12, from which *Simple 0* is excluded due to its negligible resource requirements compared to other models. Similarly, *Simple 2* is excluded because its *basic* version failed to execute on the board.

Table 3.7: Performance Evaluation of RNN Models

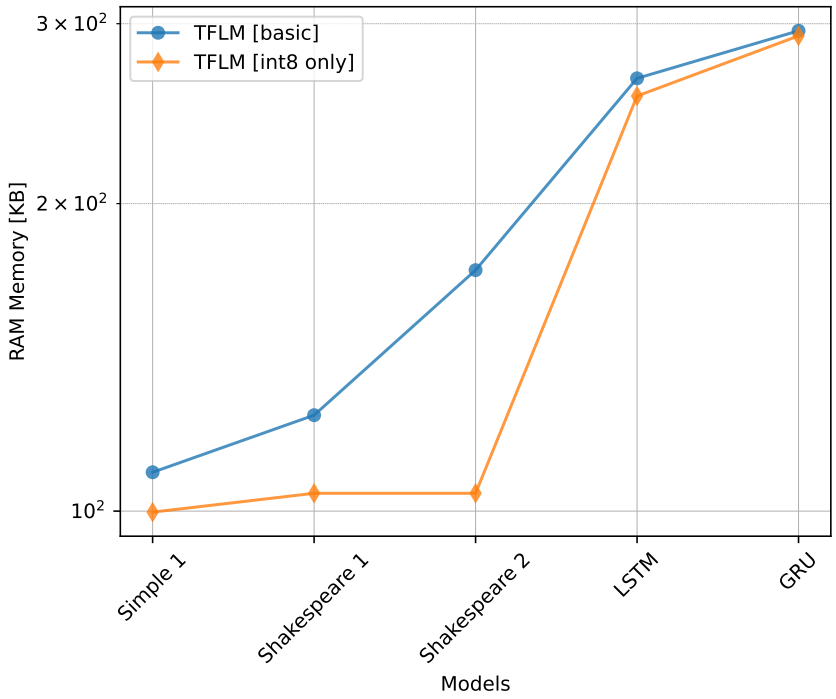
Model	**Variant**	**Error**	**Exe** (ms)	**Flash** (kB)	**RAM** (kB)
Simple 0	basic	0	0.11	110.4	8.1
	int8 only	0.006	0.14	111.6	7.1
Simple 1	basic	0	107.2	218.7	112.6
	int8 only	0.004	292.9	590.8	103.3
Simple 2	basic	-	-	-	-
	int8 only	0.004	292.9	615.2	106.4
Shakespeare 1	basic	0	141.2	251.6	127.6
	int8 only	0.021	168.4	620.8	107.6
Shakespeare 2	basic	0	377.2	348.2	175.6
	int8 only	0.022	323.5	645.2	107.6
LSTM	basic	0	362.2	473.0	268.7
	int8 only	0.014	565.3	769.8	258.4
GRU	basic	0	276.7	497.0	298.7
	int8 only	0.044	374.4	809.8	295.4

- **Model correctness:** With the exception of the *basic* version of *Simple 2* which runs out of memory, all models achieved acceptable error rates. The most concerning case is *GRU*, which may require more attention.
- **Execution time:** Surprisingly, the *basic* variants of the models show faster execution times than their *int8 only* counterparts, with the exception of *Shakespeare 2*.
- **Flash size:** Against our expectations, Fig. 3.12a reveals that the *int8 only* variants require significantly more flash memory compared to the *basic* versions.
- **RAM usage:** As shown in Fig. 3.12b, the *int8 only* variants use less RAM than *basic* models. However, we expected them to have the advantage with a bigger margin.
- **Conclusion:** This study had many unexpected results. Notably, the *int8 only* variants of the models are not as efficient as expected. The *basic* versions are faster and consume less flash memory, while they require slightly more RAM.

It is believed that LSTM is more powerful than GRU, while GRU is often favored for its computational and parameter efficiency. Consistent with this, our study observes that similar LSTMs have more parameters and a higher number of MACs compared to GRUs. However, it is interesting that using TFLM, GRUs require more flash memory and RAM than LSTMs. Despite this, GRUs still manage to execute faster than LSTMs.

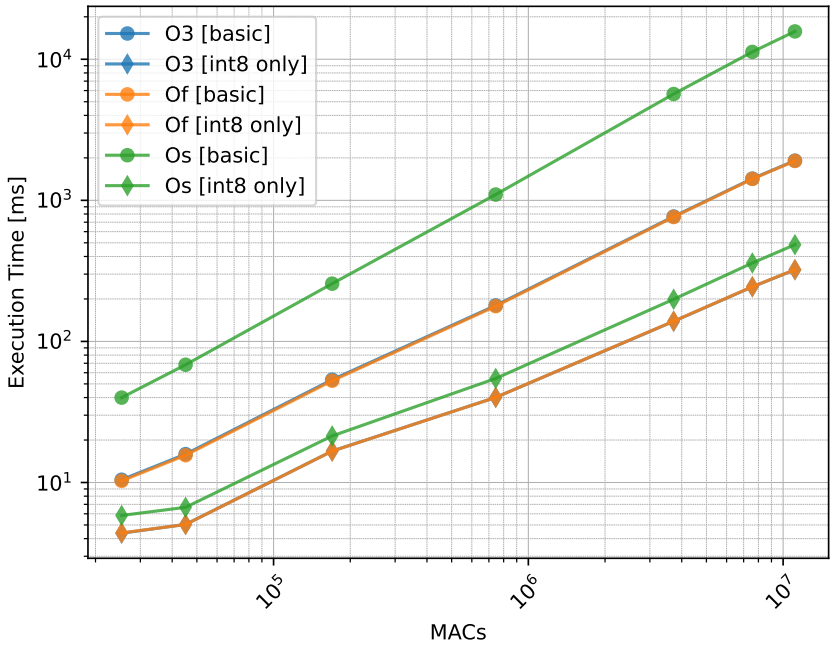


(a) Flash size.

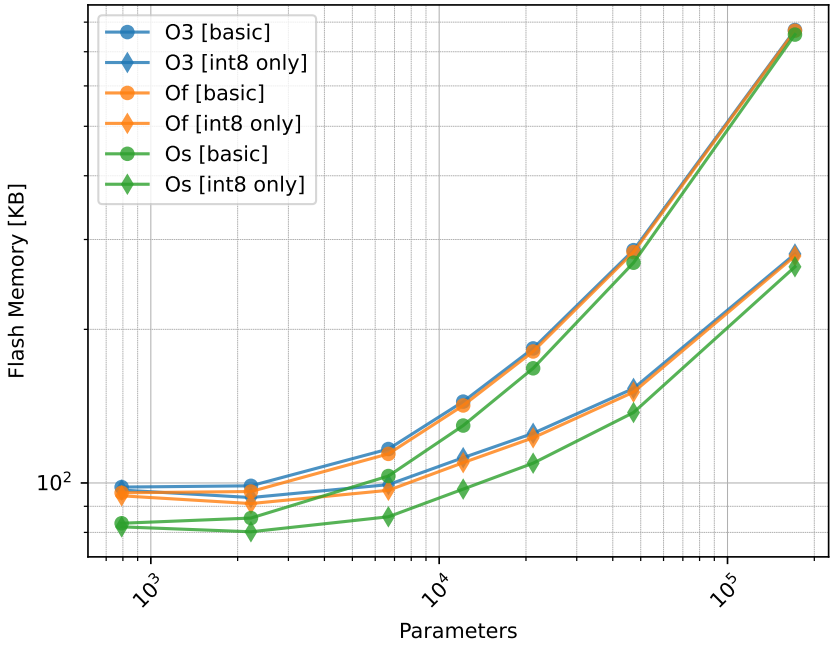


(b) RAM usage.

Figure 3.12: Performance evaluation of RNN models on the NUCLEO-L4R5ZI board.



(a) Execution time. *O3* and *Of* are almost overlapped.



(b) Flash size. *O3* and *Of* are almost overlapped.

Figure 3.13: Performance evaluation of CNN models with different GCC optimization levels on the NUCLEO-L4R5ZI board.

3.10.2 GCC Optimization Levels

The GNU Compiler Collection (GCC) is a widely used compiler for C and C++ programming languages. GCC provides several optimization levels, ranging from *O0* (no optimization) to *O3* (maximum optimization). In this study, we evaluated the impact of *Os* (optimize for size), *Of* (optimize for speed), and *O3* on the performance of FC and CNN models deployed on the NUCLEO-L4R5ZI board using TFLM.

- **Model correctness:** All three optimization levels produce the exact same outputs, confirming that the correctness of the models remains unaffected by these optimizations.
- **Execution time:** As shown in Fig. 3.13a, the *O3* and *Of* optimization levels perform almost identically and better than *Os*.
- **Flash size:** The *O3* and *Of* optimizations resulted in similar flash memory usage, while *Os* produced a slightly smaller flash size.
- **RAM usage:** No notable differences in RAM usage were observed across the three optimization levels.
- **Conclusion:** Both *O3* and *Of* provide comparable performance and outperform *Os* in terms of execution time. However, if reducing flash size is a priority, *Os* offers a slight advantage.

3.10.3 Importance of FPU

The FPU is a specialized coprocessor designed to handle floating-point arithmetic operations efficiently. To check the speedup provided by the FPU, we evaluated the performance of FC and CNN models on the NUCLEO-L4R5ZI board with and without the FPU enabled.

Our results show that FPU does not have a noticable impact on any of the evaluated metrics other than execution time. As illustrated in Fig. 3.14, the FPU provides a significant speedup (85% to 850%) for the *basic* variants of both FC and CNN models. Still, if using *int8 only*, utilizing the CMSIS-NN library can be even more beneficial and there is no need for the FPU.

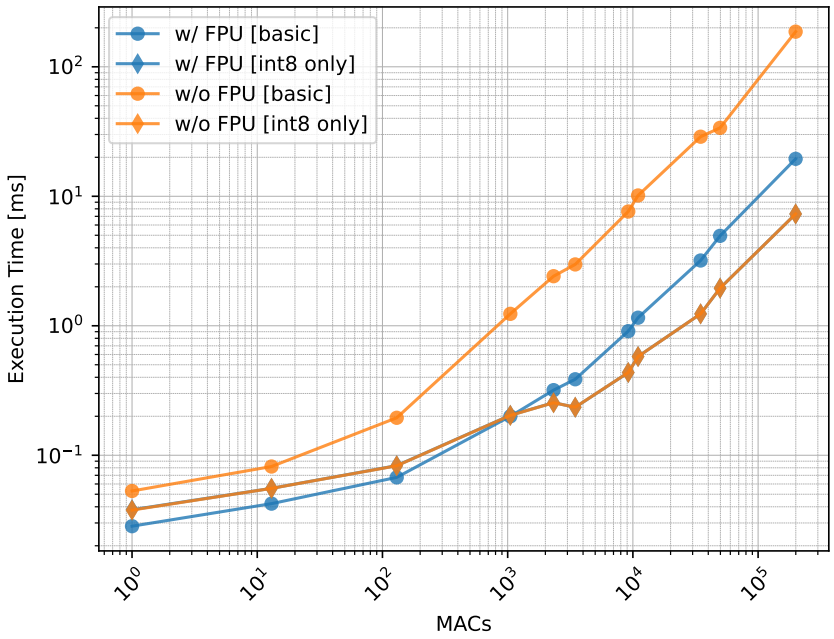
3.10.4 STM vs. Renesas

STMicroelectronics and Renesas are two major semiconductor companies that provide microcontrollers for embedded systems. STMicroelectronics is more popular between researchers and hobbyists, while Renesas is more focused on industrial and automotive applications.

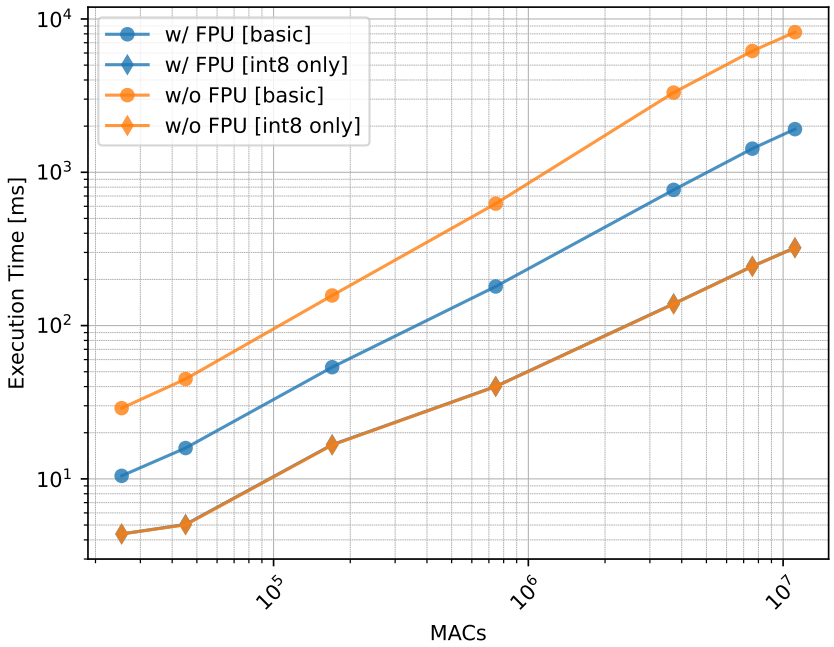
The two selected boards from these companies, STM NUCLEO-L4R5ZI and Renesas RX65N, have similar specifications, including identical flash size, RAM, and clock speed, along with many other shared features. However, the RX65N board has a proprietary RXv2 CPU architecture, while the NUCLEO-L4R5ZI board uses an ARM Cortex-M4 core. This distinction makes the comparison between these two boards particularly intriguing.

FC, CNN, RNN, and MLPerf Tiny models were evaluated on both boards using TFLM. It is worth noting that these boards have various configurable settings, which may favor one platform over the other in specific scenarios. Our study primarily relied on default settings, however it should not be considered as a comprehensive comparison between the two boards.

- **Model correctness:** The Renesas board fails to run a few of the models. Aside from this, the two boards provide similar results.

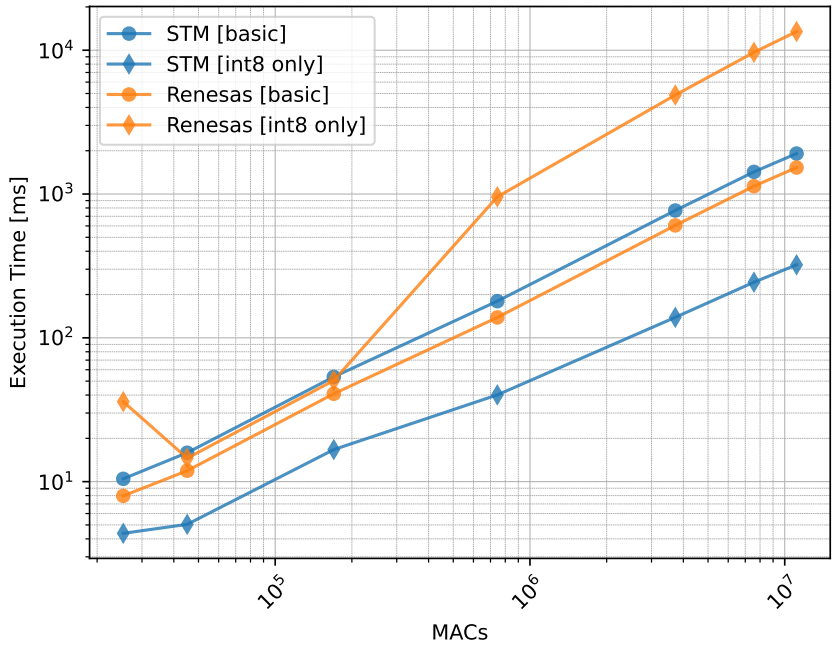


(a) FC models. *int8 only* variants are overlapped.

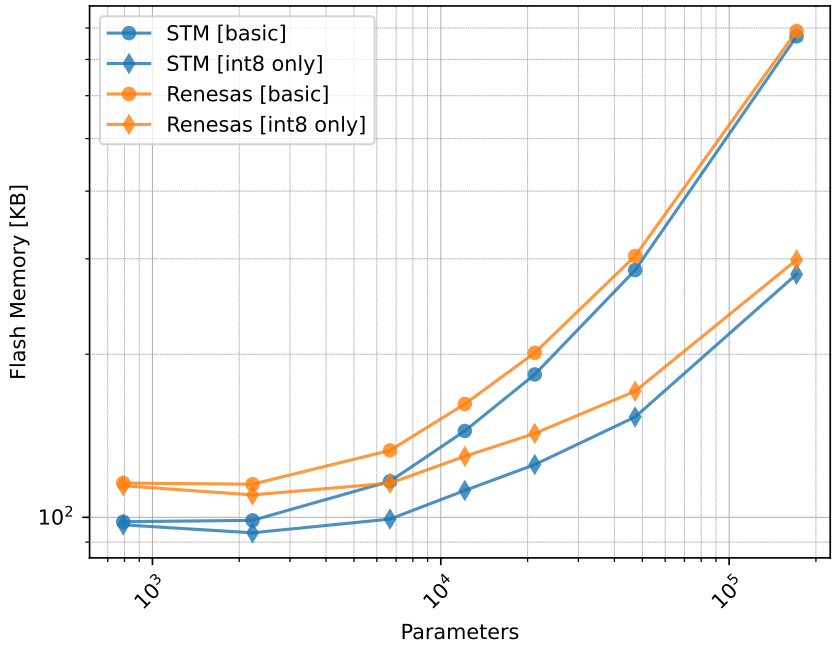


(b) CNN models. *int8 only* variants are overlapped.

Figure 3.14: Execution time of models with and without the FPU on the NUCLEO-L4R5ZI board.



(a) Execution time. The *int8 only* version of the Renesas board does not utilize CMSIS-NN and is not comparable to its STM counterpart.



(b) Flash size.

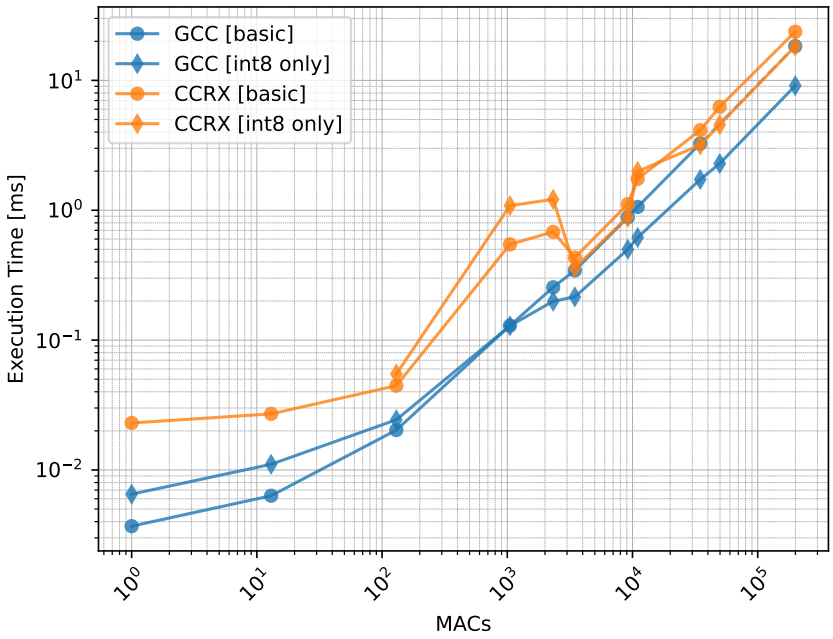
Figure 3.15: Performance evaluation of CNN models on the STM and Renesas boards.

- **Execution time:** When evaluating the *basic* versions of models and excluding *int8-only* variants (since CMSIS-NN optimizations are used only on the STM board), the Renesas board performs slightly better in terms of execution time. This is illustrated in Fig. 3.15a.
- **Flash size:** As shown in Fig. 3.15b, the STM board requires less flash memory compared to the Renesas board. The difference is more noticeable for smaller models. This is mainly due to the difference between the size of the two libraries, which becomes less significant as the model size increases.
- **RAM usage:** The RAM usage of the models is almost identical across the two boards.
- **Conclusion:** The two boards seem to have a relatively similar performance. The Renesas board might be slightly faster, but the STM board has a bit smaller flash size.

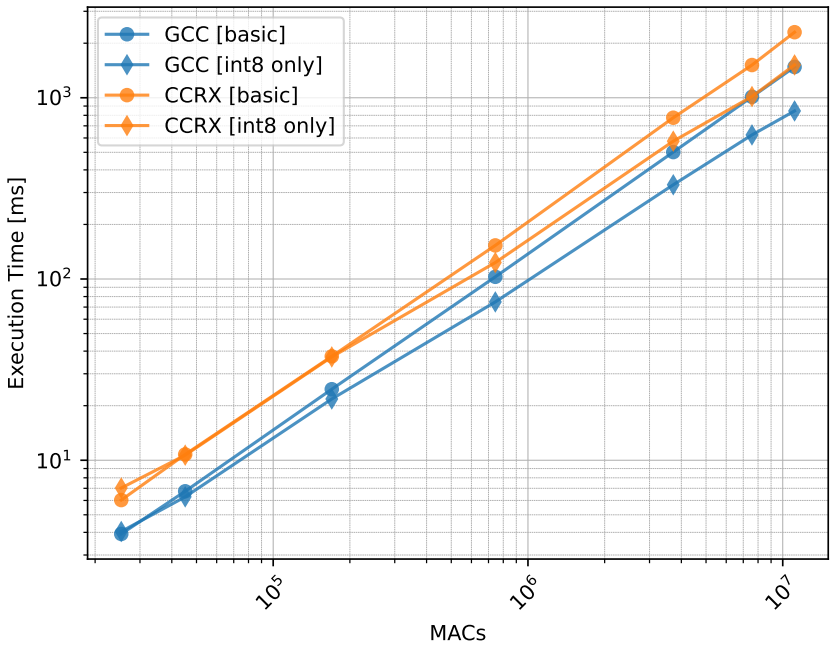
3.10.5 GCC vs CCRX

Renesas offers GCC as a free compiler for their microcontrollers, while they also provide their proprietary compiler, CC-RX, which is further optimized for RX microcontrollers. In this study, we evaluated the performance of FC and CNN models deployed on the Renesas RX65N board using eAI Translator and CC-RX and GCC compilers.

Since we could not interpret the memory requirements of the models using CC-RX, we only evaluated the execution time of the models. As evidenced by Fig. 3.16, the GCC compiler demonstrated better performance than CC-RX in terms of execution speed under default settings and highest optimization levels.



(a) FC models.



(b) CNN models.

Figure 3.16: Execution time of models deployed using CC-RX and GCC compilers on the Renesas RX65N board.

4 Paper C: A Survey of Quantization Techniques in Embedded AI Toolchains

A Survey of Quantization Techniques in Embedded AI Toolchains[1]

Mohammad Amin Hasanpour
Technical University of Denmark (DTU)
Kgs. Lyngby, Denmark
moam@dtu.dk

Xenofon Fafoutis
Technical University of Denmark (DTU)
Kgs. Lyngby, Denmark
xefa@dtu.dk

Manuel Roveri
Politecnico di Milano
Milano, Italy
manuel.roveri@polimi.it

4.1 Paper Abstract

Quantization has become a key method for enabling deep learning (DL) inference on resource-constrained embedded systems. As the demand for privacy-preserving, low-latency, and energy-efficient artificial intelligence (AI) increases, quantization allows models to run efficiently on edge hardware by reducing the precision of weights and activations—often with minimal impact on accuracy. This survey presents a tool-centric analysis of quantization support in twelve widely used embedded artificial intelligence (eAI) frameworks, including TensorFlow Lite, PyTorch, ONNX Runtime, and vendor-specific stacks like Qualcomm's QNN and Intel's OpenVINO. We examine how each tool implements quantization across several axes: supported workflows (post-training vs. quantization-aware training), bit-width flexibility, execution realism (simulated vs. integer kernels), and quantization granularity and schemes. Our findings reveal common patterns—such as the dominance of 8-bit uniform affine quantization—and highlight key distinctions in flexibility, deployment readiness, and hardware integration. We summarize our results in a unified comparison table to guide practitioners and researchers in selecting the most appropriate tool for their deployment needs. Finally, we discuss trends such as mixed-precision quantization and speculate on future directions for eAI tooling.

Index Terms—Quantization, TinyML, Embedded AI, Machine Learning, Deep Learning

4.2 Introduction

AI is transforming sectors like healthcare, automotive, and consumer electronics. While cloud infrastructure supports complex DL models, the need for real-time, private, and energy-efficient inference is driving AI to the edge.

[1]This work is supported by the Innovation Fund Denmark for the project DIREC (9142-00001B). Additional support for the research stay at Politecnico di Milano was provided by the Danish Society of Engineers (IDA) and Thomas B. Thriges Fond.

Embedded artificial intelligence (eAI), often referred to as tiny machine learning (TinyML), enables machine learning (ML) on resource-limited embedded devices. This approach reduces dependence on internet connectivity, enhances data privacy, lowers energy use, and supports low-latency applications like gesture recognition and anomaly detection.

To enable the deployment of DL models on constrained hardware, various model compression and optimization techniques have been proposed, including pruning, knowledge distillation, architecture search, and quantization. Among these, quantization has become especially prominent due to its simplicity, effectiveness, and broad support across frameworks and hardware platforms. By reducing the numerical precision of weights and activations, quantization not only reduces model size but also enables efficient integer arithmetic, leading to faster and more power-efficient inference.

This survey focuses on how quantization is supported and implemented in state-of-the-art eAI toolchains. We investigate twelve prominent toolsets, ranging from inference engines and quantization libraries to compression frameworks, highlighting their approaches to quantizing neural networks for deployment on edge hardware. Specifically, we examine what components are quantized, which quantization strategies (e.g., post-training vs. quantization-aware training) are supported, how fine-grained the configuration is (e.g., per-layer bit-widths), whether they simulate or perform real quantized execution, and which quantization schemes and bit-widths are available.

4.2.1 Scope and Contributions

The key contributions of this survey are:

- We thoroughly review twelve widely-used eAI tools, spanning frameworks like PyTorch, TensorFlow, ONNX, and OpenVINO, with emphasis on their quantization capabilities.
- We offer a comparative analysis of these tools across critical dimensions, such as supported quantization types, hardware compatibility, and work flows.
- We summarize our findings in a unified table that enables practitioners to quickly identify the right tool for their application needs.
- We highlight emerging trends and trade-offs in embedded quantization and possible directions for the future of eAI tools.

4.2.2 Organization of the Paper

The remainder of this paper is organized as follows. Section 4.3 reviews prior work related to TinyML and quantization. Section 4.4 provides background on quantization fundamentals, including common schemes and deployment implications. Section 4.5 describes the methodology used in selecting and analyzing the tools. Section 4.6 presents a detailed survey of quantization support in eAI toolchains. Section 4.7 discusses trends, trade-offs, provides practical recommendations, and outlines directions for future research. Finally, Section 4.8 concludes the paper.

4.3 Related Work

4.3.1 TinyML Surveys

Several recent surveys provide broad overviews of TinyML, its workflows, and applications on resource-constrained devices. [110] delivers a taxonomy of TinyML techniques and use cases, ranging from environmental sensing to anomaly detection, and emphasizes the critical role of model compression methods such as quantization for enabling sub-milliwatt inference on microcontrollers. [111] conducts a PRISMA-driven review that

categorizes TinyML workflows into ML-oriented, hardware-oriented, and co-design approaches; it covers model optimization methods (pruning, quantization, distillation) alongside state-of-the-art TinyML hardware and software stacks. [112] focuses on “reformable” TinyML (i.e. enabling post-deployment updates) and surveys the available toolchains and benchmarks, noting that quantization remains a foundational technique for fitting deep networks in on-device memory. [113] underscores that the extreme memory and energy constraints on MCUs make quantization, alongside pruning and distillation, indispensable for practical deployment.

Many of these surveys include eAI tools as part of their study and provide a general description of them. In particular, [114] explores different levels of the TinyML stack, with toolchains being a part of it. It offers an overview of many eAI tools, focusing on both traditional ML and neural network (NN) algorithms. [115] delves further into the study of these tools and provides a framework for automating their usage and benchmarking them.

4.3.2 Quantization Surveys

General surveys on low-precision neural networks chart the evolution of quantization methods from binary/ternary schemes to modern mixed-precision and learnable quantizers. [116] outlines the theoretical foundations of quantized NNs and reviews their hardware implications, arguing that quantization yields orders-of-magnitude savings in model size and energy consumption. [117] performs an empirical evaluation of post-training quantization (PTQ) algorithms for TinyML, benchmarking sub-8-bit quantization schemes on representative microcontroller workloads and highlighting the accuracy-memory trade-offs inherent to ultra-low-bit quantization. [118] presents MicroAI, an end-to-end framework for training, quantizing (8-bit and 16-bit), and deploying DNNs on 32-bit Cortex-M MCUs; they compare its performance and memory footprint against TensorFlow Lite Micro and STM32Cube.AI, demonstrating that quantized models can meet stringent on-device constraints without significant loss in accuracy.

4.3.3 Tool-Centric Surveys

While numerous toolkits support quantization, there is limited literature that systematically compares these frameworks in the context of eAI deployment. Existing work tends to focus on individual tool capabilities or case-study benchmarks. For instance, several “how-to” guides and whitepapers describe quantization flows in TensorFlow Lite (TFLite), PyTorch, and Open Neural Network Exchange (ONNX) Runtime, but they lack comprehensive, side-by-side analyses of bit-width flexibility, real vs. simulated kernels, and per-layer configurability. To our knowledge, no prior survey aggregates quantization features across both research-oriented libraries and industry-grade inference engines. This gap motivates our work: a unified, tool-centric survey that delineates how each eAI framework implements quantization, covering parameter vs. activation coverage, post-training quantization (PTQ) vs. quantization-aware training (QAT) support, mixed-precision capabilities, simulated vs. hardware-native operations, and supported quantization schemes and bit-widths.

4.4 Background: Quantization Fundamentals

Quantization is a model compression technique that reduces the numerical precision of neural network weights and activations, mapping high-precision values (e.g., 32-bit floats) to lower-precision representations (e.g., 8-bit or 4-bit integers). This reduces the model’s memory footprint and computational cost (enabling fast, low-power inference on edge hardware) at the expense of introducing *quantization error* from rounding and clipping.

In practice, quantization is often formulated as a linear (uniform) mapping: a real value x

is converted to an integer grid point via

$$x_{\text{int}} = \text{round}(x/s) + z, \tag{4.1}$$

where the **scale** s (step size) and **zero-point** z are chosen per tensor or channel to cover the desired dynamic range [119]. The integer result x_{int} is then constrained to a fixed-width representation, typically within a range determined by the target **bit-width** (e.g., $[0, 255]$ for unsigned 8-bit). When values fall outside this representable range, they are clipped to the nearest bound—a process known as **clipping**. The dequantized value is then given by

$$\hat{x} = s \cdot (x_{\text{int}} - z). \tag{4.2}$$

This affine quantization scheme allows efficient fixed-point arithmetic. A common simplification is **symmetric quantization**, where $z = 0$ and the range is centered on zero, reducing overhead at the cost of limited range coverage [119].

We summarize several common quantization schemes:

- **Uniform (affine) quantization:** Uses a constant step size s and zero-point z to linearly map floats to an integer range. The zero-point ensures that real zero is represented exactly. Most 8-bit quantizers (e.g., in TensorFlow Lite (TFLite) or PyTorch) use this scheme. The special case of **symmetric** quantization ($z = 0$) simplifies hardware but may reduce dynamic range [119].
- **Power-of-two quantization:** A symmetric quantizer in which the scale s is constrained to a power-of-two ($s = 2^{-k}$), so scaling is implemented by bit shifts [119]. This enables extremely efficient integer arithmetic but restricts the available scaling granularity.
- **Non-uniform quantization:** Employs nonlinear mappings (e.g., logarithmic or clustering-based) to better match the distribution of values. These schemes can reduce quantization error for certain data distributions, but are less common due to higher complexity and limited framework support.
- **Extreme low-bit quantization (binary/ternary):** Constrains weights and/or activations to very small discrete sets, such as $\{+1, -1\}$ (binary) or $\{+1, 0, -1\}$ (ternary). These schemes, as used in binary neural networks (BNNs) and ternary neural networks (TNNs), offer extreme compression but require specialized training techniques (e.g., straight-through estimators) to preserve model accuracy. These schemes require special attention for implementation, since standard processors are not designed for sub-8-bit arithmetic.

The process of uniform affine quantization is illustrated in Figure 4.1. It shows how continuous real-valued inputs are mapped to discrete integer levels using a linear scaling factor and an offset (zero-point). This example features 3-bit quantization, yielding eight quantization bins. As shown, real values (green) are aligned to their closest quantization level (blue ticks), producing integer representations. Such visualizations help clarify how quantization discretizes the input space and where quantization error arises due to rounding and clipping.

Quantization schemes can also be classified by **granularity**:

- **Per-tensor quantization** applies a single scale and zero-point to an entire tensor.
- **Per-channel quantization** assigns distinct scales to each output channel (typically for weights), significantly improving accuracy with minimal hardware overhead.

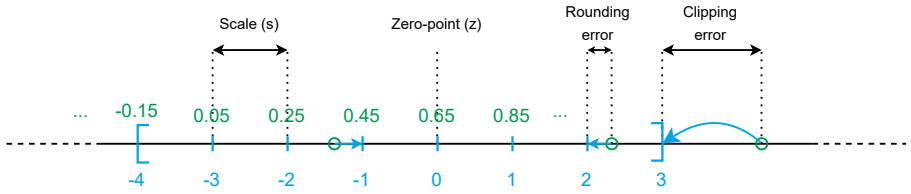


Figure 4.1: Illustration of uniform affine quantization. Green values represent real numbers, green circles mark example real values, blue ticks indicate quantization bins, and blue arrows show how these values are quantized. The quantization is asymmetric ($z \neq 0$), and the bitwidth is 3, resulting in 2^3 quantization bins.

- **Per-block quantization** goes beyond per-channel quantization by organizing weights into more complex blocks. This approach can enhance accuracy but introduces additional complexity, making it less common.

Another key axis is **precision**. While most toolchains target 8-bit precision, 4-bit (or lower) quantization is becoming more common. Some frameworks also support reduced-precision floating-point formats (e.g., 16-bit floating point (FP16), bfloat16), though these are generally distinct from integer quantization.

Two primary workflows are employed:

- **PTQ:** The model is fully trained in high precision, then quantized afterward. This may involve a calibration step on sample data to determine appropriate scales, but does not involve further training. PTQ is simple and efficient, but performance can degrade if the model is not robust to quantization noise.
- **QAT:** Quantization effects are simulated during training so the model can adapt. Fake quantization operators (quantize-dequantize) are inserted in the forward path, while gradients are passed through using a straight-through estimator [120]. Although having the same memory and compute requirement as PTQ, QAT typically yields better accuracy, especially for low-bit precision, at the cost of additional training effort.

Another important distinction is between simulated (fake) quantization and real quantization. In fake quantization, quantization effects are emulated during training or calibration by inserting quantize-dequantize (QDQ) operations into the computation graph. The model continues to operate on high-precision values, but the inserted operations mimic quantization behavior to expose the network to potential information loss. This is common during the training in the QAT workflow. By contrast, real quantization refers to models that are numerically quantized (with integer weights and activations) and executed using low-precision integer kernels on the target hardware. It is worth noting that in our discussion, by “fake PTQ”, we do not mean that the tool cannot export a quantized model, but it lacks specific kernels to efficiently execute the quantized model on any hardware.

A complementary technique is **mixed-precision quantization**, where different layers or tensors use different bit widths. Since not all layers or tensors are equally sensitive to quantization, one can use higher precision (e.g., 16-bit) for critical layers and lower precision (e.g., 8-bit or 4-bit) for others. Mixed precision thus balances efficiency and accuracy by allocating resources where they are most needed [121]. Modern toolchains increasingly support per-layer (or even per-tensor) precision configuration, enabling such

fine-grained quantization.

In summary, quantization offers a trade-off between computational efficiency and model accuracy. The choice of scheme (e.g., uniform vs. power-of-two), precision (e.g., 8-bit vs. 4-bit), and training approach (PTQ vs. QAT) must be tuned based on hardware targets, toolchain support, and application-specific accuracy requirements.

4.5 Methodology

Our survey aims to provide a structured and tool-centric comparison of how quantization is implemented across eAI frameworks. To this end, we adopted a qualitative, documentation-driven methodology grounded in publicly available resources and hands-on inspection.

4.5.1 Tool Selection

We selected twelve widely-used eAI tools that either (i) focus explicitly on quantization, or (ii) offer extensive quantization support as part of their deployment or optimization pipeline. These tools are: TFLite, PyTorch Quantized Neural Network PACKage (QNNPACK), PyTorch Quantization, QKeras, Larq, Plumerai, Brevitas, Neural Network Intelligence (NNI), QNN Model Porting, ONNX Runtime, AI Model Efficiency Toolkit (AIMET), and Neural Network Compression Framework (NNCF). They span a range of categories, including general-purpose deep learning frameworks, research-oriented quantization libraries, and vendor-specific deployment toolchains. Our goal was not to exhaustively cover all quantization-capable libraries, but rather to focus on those with significant usage in embedded inference and quantization-centric design.

4.5.2 Data Collection and Analysis

To characterize each tool's quantization support, we collected information across several key dimensions:

- What model components can be quantized (weights, activations, biases).
- Whether the tool supports PTQ, QAT, or both.
- Whether per-layer or mixed-precision configurations are possible.
- Whether quantization is simulated (fake quantization) or realized through integer operations on target hardware.
- Supported quantization bit-widths
- Supported quantization schemes (e.g., uniform, symmetric, binary).

Our primary data source was each tool's official documentation, including tutorials, specification pages, and technical references. When documentation was incomplete or ambiguous, we consulted additional resources such as open-source repositories, API definitions, code comments, and blog posts from tool maintainers. In some cases, we inferred behavior from usage patterns or implementation details when direct confirmation was unavailable.

4.5.3 Limitations

Table 4.1 details the current state of the tools at the time of writing, on which our analysis is based. Because some tools evolve rapidly, especially in terms of quantization features and hardware backend support, specific capabilities may change over time. Additionally, while we made a concerted effort to verify the accuracy of our findings, some interpretations, particularly those based on source code, may be incomplete or imprecise due to limited or undocumented behavior.

Table 4.1: The status of the studied tools at the time of writing. Plumerai and QNN Model Porting were not open-source, so we have excluded them from the table. Information for these and other tools was collected in July 2025. The number of *GitHub stars* in parentheses indicates that the tool is part of a larger project, and the star count corresponds to that project. The *Last update* column shows how many months have passed since the last commit of that tool on GitHub.

Tool	Version	GitHub stars	Last update
TFLite [122]	2.19.0	(191k)	<1
PyTorch QNNPACK [123]	N/A	1.5k	71
PyTorch Quantization [124]	2.7.1	(91.7k)	4
QKeras [125]	0.9.0	566	1
Larq [126]	0.13.3	719	11
Brevitas [127]	0.12.0	1.4k	2
NNI [128]	3.0	14.2k	21
ONNX Runtime [129]	1.22.1	17.3k	<1
AIMET [130]	2.10.0	2.4k	<1
NNCF [131]	2.17.0	1.1k	<1

Despite these limitations, we believe the results offer a representative and practically useful overview of quantization practices across modern eAI toolchains.

It is worth noting that EdgeMark [115] offers a unified framework for supporting and benchmarking a wide range of eAI tools. In future work, we aim to extend its support for quantization methods to enable broader quantization-focused comparisons.

4.6 Quantization in Embedded AI Toolchains

This section presents a detailed analysis of quantization support across twelve widely used eAI toolchains. For each tool, we provide a structured account of its quantization capabilities, including supported workflows (PTQ/QAT), precision and bit-width flexibility, quantization schemes, and execution behavior. A comparative summary table is provided at the end of this section (Table 4.2) to enable side-by-side comparison.

4.6.1 TensorFlow Lite

TFLite is a lightweight framework for deploying TensorFlow models on mobile and embedded devices. It supports both PTQ and QAT (the latter via the TensorFlow Model Optimization Toolkit (TFMOT)). In typical use, TFLite quantizes both weights and activations to 8-bit integers, while keeping biases in 32-bit integer format [132]. By default, a model is quantized in a single mode (e.g., full 8-bit or float16) for the entire network—mixed-precision per-layer is not generally exposed to the user. However, TFLite does allow some flexibility: for example, one can choose dynamic-range quantization (weights to 8-bit integer (INT8), activations remain float), full integer quantization (calibration-based INT8 for both), or FP16 quantization (weights stored as 16-bit floats). An experimental mode also supports 16-bit integer (INT16) activations with INT8 weights [133].

Quantization in TFLite uses a uniform affine scheme: weights are typically quantized symmetrically (zero-point at 0) on a per-channel basis for conv/dense layers, while activations are quantized asymmetrically per tensor [132]. During QAT, fake quantization operations are inserted into a Keras model so the network learns to accommodate quantization error, but the deployed model uses native integer kernels. At inference, TFLite provides real

optimized integer operators: on CPU it relies on XNNPACK or QNNPACK libraries (ARM NEON or x86 SIMD), and it can delegate execution to GPUs (via the GPU delegate), to DSPs/NPUs through Android Neural Networks API (NNAPI) or vendor delegates (e.g., Hexagon), and to specialized accelerators like the Coral Edge TPU (which requires a fully quantized 8-bit model) [134].

4.6.2 PyTorch QNNPACK

QNNPACK is an inference library in PyTorch designed for quantized NNs on mobile devices. It leverages the fact that the necessary memory block for matrix multiplication in mobile networks can fit in the L1 cache of even the weakest mobile devices, eliminating the need for repacking and transformation of data [135]. It implements optimized 8-bit integer operators (especially for convolution and linear layers) targeting ARM CPUs with NEON. In QNNPACK, both weights and activations are quantized to 8-bit integers at inference, and biases are also scaled to int32 (by combining weight and activation scales). There is no support for mixed precision or variable bitwidth per layer—QNNPACK assumes a uniform 8-bit quantization across the model. During development, one may use PyTorch's quantization simulation modules for QAT, but QNNPACK itself is purely an inference engine. The quantization scheme is uniform affine: weights are quantized per-channel (usually symmetrically) and activations per-tensor (symmetric or asymmetric).

4.6.3 PyTorch Quantization

PyTorch's built-in quantization framework provides a comprehensive suite of quantization tools for model optimization. It supports multiple workflows: dynamic PTQ (weights-only quantization with float activations, useful for LSTM/Transformer models), static PTQ (calibration-based full quantization), and QAT. The framework can quantize both weights and activations. Users configure quantization behavior via `QConfig` objects, allowing per-layer or module-level customization. By default, PyTorch quantizes to 8 bits, but it now also offers experimental support for 4-bit quantization and FP16 (especially on GPU backends). Quantization is simulated with fake-quant modules during QAT, but on supported hardware, it uses real integer arithmetic at inference. PyTorch supports multiple backends: on x86 it can use FBGEMM or oneDNN, on ARM it uses QNNPACK, and for GPU it can target TensorRT or other libraries. Both symmetric and asymmetric uniform quantization schemes are supported [136].

4.6.4 QKeras

QKeras is an extension of TensorFlow/Keras for designing low-precision NNs. It provides quantized versions of layers (e.g., `QConv2D`, `QDense`) and a wrapper for activation quantization (`QActivation`). In QKeras, weights, biases, and activations can all be quantized according to user-defined quantizers. All quantization in QKeras is done by simulated (fake) quantization during training; it is essentially a QAT workflow. The user specifies the number of bits and quantization style for each tensor (e.g., using `quantized_bits(8)` or `quantized_po2`); thus, different layers can have different bit-widths. QKeras supports uniform affine quantizers (with options for symmetric or asymmetric coding), as well as power-of-two quantization, binary, and ternary quantization modes. There are no special integer inference kernels in QKeras itself. Deployment typically involves exporting the trained model to hardware via HLS or using compatible inference engines [125].

4.6.5 Larq

Larq is a library for building and training BNNs in TensorFlow/Keras. It provides quantized layers that accept an `input_quantizer` (for activations) and a `kernel_quantizer` (for weights). Typically, Larq quantizes weights to binary (± 1) or ternary values, and can also quantize activations if desired. Biases are generally kept in full precision. Larq uses

fake quantization (often with the straight-through estimator) during training to produce low-bit networks. Its focus is on BNNs, so it includes sign-based quantizers (e.g., `SteSign`, `ApproxSign`) that yield 1-bit weights and activations. Larq also supports ternary quantizers and some multi-bit schemes (e.g., DoReFa) for mixed-precision networks [137]. For deployment, the Larq Compute Engine (LCE) provides real inference support: it packs 1-bit weights and activations and runs highly optimized binary convolution kernels on 64-bit ARM (Cortex-A) and similar processors [138].

4.6.6 Plumerai

Plumerai offers a commercial inference engine for embedded processors (Cortex-M, Cortex-A, RISC-V, etc.) with roots in binary NNs. The engine itself does not perform quantization; instead, it executes a pre-quantized model "as-is" [139]. It was originally designed for BNNs (as Larq being a part of their development system), typically with binary weights/activations except 8-bit inputs and outputs, but now also supports standard 8-bit quantized networks and even 16-bit integer arithmetic (e.g., for LSTMs). Thus, weights and activations must be quantized (to 1-, 8-, or 16-bit) prior to using Plumerai. During inference, the engine performs real integer or binary computations on the hardware, greatly reducing memory and compute cost. Plumerai provides no PTQ or QAT tools itself; users generally prepare models using TFLite, Larq, or similar, then deploy to Plumerai's runtime [140].

4.6.7 Brevitas

Brevitas is a PyTorch library (by Xilinx) for quantization research. It supplies quantized versions of common layers (e.g., `QuantConv2d`, `QuantLinear`) with configurable bit widths. Brevitas supports both PTQ and QAT. Users can choose to quantize weights, biases, and/or activations independently on each layer, with bit-widths ranging from binary (1-bit) up to 8-bit or higher [127]. Mixed-precision networks are fully supported. All quantizers in Brevitas are uniform affine with options for symmetric or asymmetric ranges [141].

4.6.8 NNI (Microsoft)

Microsoft's Neural Network Intelligence (NNI) toolkit includes a model compression suite with quantization modules. It primarily targets PyTorch (with some support for TensorFlow). NNI supports both PTQ and QAT workflows. For PTQ, it includes a "NaiveQuantizer" (often used for 8-bit weight-only quantization), while for QAT it offers algorithms like `QAT_Quantizer`, `DoReFaQuantizer`, and a `BNNQuantizer` for binarization. The user can specify which tensors to quantize (weights, activations, etc.) and the bitwidth on a per-layer basis. NNI's quantization is done via simulation in PyTorch (fake quant); it has no native integer inference engine. For deployment, one can export the (fake-quantized) model to a backend like TensorRT to run real quantized kernels [142].

4.6.9 QNN Model Porting

Qualcomm's AI Engine Direct SDK includes a QNN model converter for targeting Qualcomm hardware (CPU, GPU, or HTP). Given a pre-trained floating-point model, the converter performs PTQ only (no QAT) into low-precision formats. It can quantize both weights and activations, typically to 8-bit or 16-bit integers, and it also supports some 4-bit quantization modes. The supported quantization modes (via the `--quantize_full_type` flag) include `int8` (W/A = INT8), `int16` (W/A = INT16), `w8a16` (weights INT8, activations INT16), `w4a8` (weights INT4, activations INT8), etc. The converter outputs a model that uses real integer arithmetic on the target Qualcomm device [143].

4.6.10 ONNX Runtime

ONNX Runtime is a cross-platform inference engine that provides quantization tooling for ONNX models. Its PTQ utilities can produce quantized ONNX graphs in either QOperator

form (real operators) or with QDQ nodes. By default, ONNX quantization applies 8-bit linear (uniform affine) quantization to weights and activations (with symmetric or asymmetric coding). It also supports specialized modes such as weight-only 4-bit block-wise symmetric quantization for certain operators, and FP16 (especially on GPU). At runtime, ONNX Runtime will use hardware-accelerated integer kernels when available (for example, Intel VNNI on x86 or ARM v8.2-A dot-product instructions), or fall back to QDQ operators if not [144].

4.6.11 AIMET

AIMET is a library for quantization simulation and QAT, with a focus on Qualcomm's inference workflows. It integrates with PyTorch, TensorFlow, and ONNX. AIMET's Quantization Simulation (QuantSim) inserts fake quantize/dequantize nodes into a model to analyze the effects of PTQ. It supports standard precision modes like W8/A8, as well as mixed modes such as W8/A16 or W4/A8, and even FP16 (W-fp16/A-fp16). Users can allocate different precisions to different layers (mixed precision) to optimize accuracy. All quantization in AIMET (PTQ simulation and QAT) is done in software (fake quantization). For deployment, the resulting quantized model can be tested on real devices via Qualcomm's cloud lab or deployed locally using the AI Engine Direct SDK (as QNN Model Porting being a part of this process) [145].

4.6.12 NNCF

The NNCF is Intel's toolkit (often used with OpenVINO) for model compression, including quantization. It primarily targets INT8 quantization, but also supports other lower bit-widths (INT4, FP8, FP16, etc.) and novel formats like E2M1 (4-bit FP). NNCF offers both QAT and PTQ, as well as automated mixed-precision algorithms. Its quantization algorithms generally use uniform affine quantizers. After compression, NNCF exports the model for real inference through the OpenVINO runtime, which uses optimized integer kernels on Intel hardware [131].

4.7 Discussion and Future Directions

Our survey reveals several consistent patterns and notable differences in how quantization is implemented across eAI tools. While nearly all tools support quantization of weights and activations, their treatment of biases varies. In practice, biases are often left in higher precision formats such as int32 or float32, since their memory footprint is negligible and preserving precision can improve accuracy without incurring significant cost. For example, TFLite and PyTorch's QNNPACK store biases in int32, facilitating efficient accumulation during inference. Similarly, Larq—which targets extreme low-bit quantization—keeps biases in full precision to counteract the accuracy degradation often associated with binary or ternary weights and activations.

QAT is widely supported in many toolchains but is consistently implemented via simulated quantization using fake-quant operators. This is likely due to the fact that training typically occurs on GPUs, which do not natively support integer arithmetic in the low-precision formats used for inference. As a result, no surveyed tool uses real quantized kernels during training. PTQ, in contrast, is nearly ubiquitous and supported even in tools where QAT is the primary focus (e.g., even QKeras can do PTQ with a bit of effort). PTQ offers a fast and accessible pathway to deploy quantized models, but it can introduce unacceptable accuracy loss in models already near their capacity limits. In such cases, QAT serves as an effective remedy by enabling the model to adapt to quantization effects during training.

Real execution of quantized models using integer kernels is available in only a subset of tools, typically those oriented toward production deployment. TFLite, PyTorch (via QN-

Table 4.2: Summary of quantization capabilities of the surveyed eAI tools. The *Params* in *Quantizes* column include both weights and biases. In the *Type* column, *UA* stands for uniform affine.

Tool	**Quantizes**	**PTQ/QAT**	**Mixed precision**	**Fake/Real**	**Bits**
TFLite	Params, Acts (Biases in int-32)	Both (QAT via TF-MOT)	No	Fake QAT, Real PTQ	8-bit (f
PyTorch QNNPACK	Params, Acts (Biases in int-32)	PTQ	No	Real (ARM, Mobile devices)	8-bit
PyTorch Quantization	Params, Acts	Both	Yes	Fake QAT, Real PTQ (x86, ARM, GPU)	4-bit, (
QKeras	Params, Acts	QAT	Yes	Fake	User-d
Larq	Weights, Acts	Both	Yes	Fake QAT, Real PTQ via LCE (ARM)	Binary,
Plumerai	Params, Acts	N/A	N/A	Real (ARM, etc)	1-bit, 8
Brevitas	Params, Acts	Both	Yes	Fake	Config
NNI	Params, Acts	Both	Yes	Fake	Config
QNN Model Porting	Params, Acts	PTQ	No	Real (Qualcomm)	4-bit, 8
ONNX Runtime	Params, Acts	PTQ	No	Real on supported hardware	4-bit, 8
AIMET	Params, Acts	Both	Yes	Fake	FP16, W8/A8
NNCF	Params, Acts	Both	Yes	Fake QAT, Real PTQ (via OpenVINO)	8-bit, (

NPACK), ONNX Runtime, Qualcomm's QNN SDK, and Plumerai all offer such support, often leveraging optimized kernels for specific hardware. Many research-oriented tools, on the other hand, focus on training and simulation workflows, deferring actual deployment to external backends.

In terms of bit-width, 8-bit quantization stands out as the industry standard for its balance of accuracy, memory efficiency, and hardware support. It is the most widely supported bit-width across surveyed tools. Other common formats include 16-bit (e.g., FP16 for GPUs) and 4-bit quantization for further compression. Only a few tools—such as QKeras and Brevitas—allow arbitrary bit-widths, making them particularly attractive for experimentation, though these often require custom deployment. Binary and ternary quantization remain niche; even Plumerai, once focused on BNNs and TNNs, now supports 8- and 16-bit formats, reflecting limited readiness of ultra-low-bit methods for general use.

Uniform affine quantization is by far the dominant scheme across all tools. It is simple, efficient, and enjoys broad hardware support. A common pattern is to quantize weights symmetrically and activations asymmetrically, which balances representation accuracy with computational efficiency during inference. Non-uniform or nonlinear quantization schemes, such as logarithmic or clustering-based methods, may better match certain data distributions but are not widely adopted due to their complexity and poor hardware support. Binary and ternary schemes are treated as special cases and require distinct training methods and inference backends.

For practitioners, tool selection is strongly influenced by the deployment target and required feature set. Production-focused engineers benefit most from frameworks that support real inference with optimized integer kernels, such as TFLite, ONNX Runtime, and vendor-specific SDKs. Researchers and experimentalists may prefer flexible tools such as QKeras, which allow fine-grained control over bit-widths, quantization styles, and training workflows. Table 4.2 provides a comparison that can guide users in matching tool capabilities to project requirements.

Despite the diversity of tools and devices, quantization itself is largely hardware-agnostic. A standard like 8-bit uniform affine quantization serves as a "push-button" compression technique applicable across platforms, providing a compelling case for standardizing quantization as a distinct preprocessing step. Such unification would allow model developers to quantize once and deploy across a variety of hardware backends, each executing the quantized model using its own optimized kernels.

Looking ahead, several trends point to the increasing relevance of mixed-precision quantization. As AI accelerators evolve to support various bit lengths and as quantization algorithms become more sophisticated, mixed-precision workflows are poised to offer new levels of efficiency without compromising accuracy. Some tools already support mixed-precision training or export, and broader support is expected to follow. We anticipate that mixed-precision quantization will become more popular in the future, particularly as the boundaries between training, quantization, and deployment become more fluid and tightly integrated.

4.8 Conclusion

Quantization has emerged as a key enabler for deploying DL models on resource-constrained embedded systems, striking a balance between accuracy, model size, and computational efficiency. In this survey, we provided a structured overview of how quantization is implemented across twelve widely-used eAI toolchains, encompassing both research-focused libraries and production-grade inference engines.

Our analysis highlights the widespread adoption of uniform affine quantization—particularly 8-bit integer formats—as a practical standard across platforms. While support for PTQ is nearly universal, QAT is increasingly adopted to preserve model accuracy, especially under aggressive quantization regimes. We also observe a growing interest in mixed-precision and ultra-low-bit quantization, though their deployment is currently limited to specialized tools or custom hardware targets like FPGAs.

Despite the diversity of tooling, a clear pattern emerges: training and quantization are typically decoupled from actual integer execution. This reinforces the importance of tool interoperability and standardized quantization formats to bridge development and deployment pipelines.

As eAI continues to mature, we expect future work to focus on tighter integration between training, quantization, and hardware execution. This includes expanding support for mixed-precision strategies, advancing hardware-aware quantization algorithms, and enabling model compression flows that are both transparent and hardware-adaptive. Ultimately, quantization will remain central to the design of efficient, privacy-preserving, and low-latency AI systems at the edge.

5 Conclusions and Future Directions

Conclusion goes here.

Bibliography

[1] Maxime Binama, Alex Muhirwa, and Emmanuel Bisengimana. *Cavitation Effects in Centrifugal Pumps-A Review*. 2016.

[2] Christian Brix Jacobsen. *The Centrifugal Pump*. 1st ed. Grundfos Management A/S, Sept. 2008.

[3] Ahmed Ramadhan Al-Obaidi. "Detection of Cavitation Phenomenon within a Centrifugal Pump Based on Vibration Analysis Technique in both Time and Frequency Domains". In: *Experimental Techniques* 44 (3 2020), pp. 329–347. issn: 17471567. doi: 10.1007/s40799-020-00362-z.

[4] Yi Li et al. "An experimental study on the cavitation vibration characteristics of a centrifugal pump at normal flow rate". In: *Journal of Mechanical Science and Technology* 32 (10 Oct. 2018), pp. 4711–4720. issn: 1738494X. doi: 10.1007/s12206-018-0918-x.

[5] Ahmed Ramadhan AL-OBAIDI. "Experimental comparative investigations to evaluate cavitation conditions within a centrifugal pump based on vibration and acoustic analyses techniques". In: *Archives of Acoustics* 45 (3 2020), pp. 541–556. issn: 2300262X. doi: 10.24425/aoa.2020.134070.

[6] A. M. Abdulaziz and Ashraf Kotb. "Detection of pump cavitation by vibration signature". In: *Australian Journal of Mechanical Engineering* 15 (2 May 2017), pp. 103–110. issn: 14484846. doi: 10.1080/14484846.2015.1093261.

[7] Ruijia Cao et al. "Numerical method to predict vibration characteristics induced by cavitation in centrifugal pumps". In: *Measurement Science and Technology* 32 (11 Nov. 2021). issn: 13616501. doi: 10.1088/1361-6501/ac1181.

[8] Georgios Mousmoulis et al. "Experimental analysis of cavitation in a centrifugal pump using acoustic emission, vibration measurements and flow visualization". In: *European Journal of Mechanics, B/Fluids* 75 (May 2019), pp. 300–311. issn: 09977546. doi: 10.1016/j.euromechflu.2018.10.015.

[9] Mohammad Taghi Shervani-Tabar et al. "Cavitation intensity monitoring in an axial flow pump based on vibration signals using multi-class support vector machine". In: *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science* 232 (17 Sept. 2018), pp. 3013–3026. issn: 20412983. doi: 10.1177/0954406217729416.

[10] Ali Hajnayeb. "Cavitation Analysis in Centrifugal Pumps Based on Vibration Bispectrum and Transfer Learning". In: *Shock and Vibration* 2021 (2021). issn: 10709622. doi: 10.1155/2021/6988949.

[11] Nabanita Dutta et al. "Centrifugal Pump Cavitation Detection Using Machine Learning Algorithm Technique". In: *International Conference on Environment and Electrical Engineering (EEEIC)* (2018).

[12] Marios Karagiovanidis et al. "Early Detection of Cavitation in Centrifugal Pumps Using Low-Cost Vibration and Sound Sensors". In: *Agriculture (Switzerland)* 13 (8 Aug. 2023). issn: 20770472. doi: 10.3390/agriculture13081544.

[13] Seyed M. Matloobi and Mohammad Riahi. *Identification of cavitation in centrifugal pump by artificial immune network*. Dec. 2021. doi: 10.1177/09544089211028402.

[14] Mohammad Amin Hasanpour. *Cavitation*. 2024. url: https://github.com/Black3rror/Cavitation.

[15] Jair Cervantes et al. “A comprehensive survey on support vector machine classification: Applications, challenges and trends”. In: *Neurocomputing* 408 (Sept. 2020), pp. 189–215. issn: 18728286. doi: 10.1016/j.neucom.2019.10.118.

[16] Emil Njor, Jan Madsen, and Xenofon Fafoutis. “Data Aware Neural Architecture Search”. In: *TinyML Research Symposium* (Apr. 2023). url: https://arxiv.org/abs/2304.01821.

[17] Atis Elsts et al. “On-board feature extraction from acceleration data for activity recognition”. In: *EWSN’18: Proceedings of the 2018 International Conference on Embedded Wireless Systems and Networks*. ACM. 2018, pp. 163–186.

[18] Robert David et al. “Tensorflow lite micro: Embedded machine learning for tinyml systems”. In: *Proc. Machine Learning and Systems* 3 (2021), pp. 800–811.

[19] Liangzhen Lai, Naveen Suda, and Vikas Chandra. “CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs”. In: *arXiv preprint arXiv:1801.06601* (Jan. 2018). url: http://arxiv.org/abs/1801.06601.

[20] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. “Neural Architecture Search: A Survey”. In: *Journal of Machine Learning Research* 20 (2019), pp. 1–21.

[21] Hanxiao Liu, Karen Simonyan, and Yiming Yang. “DARTS: Differentiable Architecture Search”. In: *arXiv preprint arXiv:1806.09055* (June 2018). url: http://arxiv.org/abs/1806.09055.

[22] Han Cai, Ligeng Zhu, and Song Han. “ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware”. In: *arXiv preprint arXiv:1812.00332* (Dec. 2018).

[23] Han Cai et al. “Once-for-All: Train One Network and Specialize it for Efficient Deployment”. In: *arXiv preprint arXiv:1908.09791* (Aug. 2019). url: http://arxiv.org/abs/1908.09791.

[24] Zhuang Liu et al. “Learning Efficient Convolutional Networks through Network Slimming”. In: *IEEE international conference on computer vision* (2017), pp. 2736–2744.

[25] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. “Distilling the Knowledge in a Neural Network”. In: *arXiv preprint arXiv:1503.02531* (Mar. 2015). url: http://arxiv.org/abs/1503.02531.

[26] Colby Banbury et al. “MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers”. In: *Proc machine learning and systems* 3 (Oct. 2021), pp. 517–532. url: http://arxiv.org/abs/2010.11267.

[27] Nikhil P Ghanathe and Steve Wilton. “T-RECX: Tiny-Resource Efficient Convolutional neural networks with early-eXit”. In: *Proc. ACM International Conference on Computing Frontiers* (July 2023), pp. 123–133.

[28] Vasileios Tsoukas et al. “A Review on the emerging technology of TinyML”. In: *ACM Computing Surveys* 56 (10 Oct. 2024), pp. 1–37. issn: 0360-0300. doi: 10.1145/3661820.

[29] Riku Immonen and Timo Hämäläinen. “Tiny Machine Learning for Resource-Constrained Microcontrollers”. In: *Journal of Sensors* 2022 (Nov. 2022), pp. 1–11. issn: 1687-7268. doi: 10.1155/2022/7437023.

[30] Emil Njor et al. “A Holistic Review of the TinyML Stack for Predictive Maintenance”. In: *IEEE Access* (Dec. 2024), pp. 1–1. issn: 2169-3536. doi: 10.1109/ACCESS.2024.3512860.

[31] Ramon Sanchez-Iborra and Antonio F. Skarmeta. “TinyML-Enabled Frugal Smart Objects: Challenges and Opportunities”. In: *IEEE Circuits and Systems Magazine* 20 (3 Aug. 2020), pp. 4–18. issn: 1531-636X. doi: 10.1109/MCAS.2020.3005467.

[32] Partha Pratim Ray. “A review on TinyML: State-of-the-art and prospects”. In: *Journal of King Saud University - Computer and Information Sciences* 34 (4 Apr. 2022), pp. 1595–1623. issn: 13191578. doi: 10.1016/j.jksuci.2021.11.019.

[33] Swapnil Sayan Saha, Sandeep Singh Sandha, and Mani Srivastava. “Machine Learning for Microcontroller-Class Hardware: A Review”. In: *IEEE Sensors Journal* 22 (22 Nov. 2022), pp. 21362–21390. issn: 1530-437X. doi: 10.1109/JSEN.2022.3210773.

[34] Luigi Capogrosso et al. “A Machine Learning-Oriented Survey on Tiny Machine Learning”. In: *IEEE Access* 12 (2024), pp. 23406–23426. issn: 2169-3536. doi: 10.1109/ACCESS.2024.3365349.

[35] Youssef Abadade et al. “A Comprehensive Survey on TinyML”. In: *IEEE Access* 11 (2023), pp. 96892–96922. issn: 2169-3536. doi: 10.1109/ACCESS.2023.3294111.

[36] Colby R. Banbury et al. “Benchmarking TinyML Systems: Challenges and Direction”. In: *ArXiv* (Mar. 2020).

[37] Anas Osman et al. “TinyML Platforms Benchmarking”. In: *Applications in Electronics Pervading Industry, Environment and Society*. Ed. by Sergio Saponara and Alessandro De Gloria. Cham: Springer International Publishing, 2022, pp. 139–148. isbn: 978-3-030-95498-7.

[38] Lars Wulfert et al. “AIfES: A Next-Generation Edge AI Framework”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 46 (6 June 2024), pp. 4519–4533. issn: 0162-8828. doi: 10.1109/TPAMI.2024.3355495.

[39] Philipp van Kempen et al. “MLonMCU: TinyML Benchmarking with Fast Retargeting”. In: *Proceedings of the 2023 Workshop on Compilers, Deployment, and Tooling for Edge AI*. ACM, Sept. 2023, pp. 32–36. isbn: 9798400703379. doi: 10.1145/3615338.3618128.

[40] Vlad-Eusebiu Baciu, Johan Stiens, and Bruno da Silva. “MLino bench: A comprehensive benchmarking tool for evaluating ML models on edge devices”. In: *Journal of Systems Architecture* 155 (Oct. 2024), p. 103262. issn: 13837621. doi: 10.1016/j.sysarc.2024.103262.

[41] Robert David et al. “TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems”. In: *Proceedings of Machine Learning and Systems 3 (MLSys 2021)* (Oct. 2020).

[42] Shawn Hymel et al. *Edge Impulse: An MLOps Platform for Tiny Machine Learning*. 2023. arXiv: 2212.03332 [cs.DC]. url: https://arxiv.org/abs/2212.03332.

[43] Edgeimpulse. *Layers*. accessed: 3 April, 2025. url: https://docs.edgeimpulse.com/docs/concepts/machine-learning/neural-networks/layers.

[44] Edgeimpulse. *Introducing EON: Neural Networks in Up to 55% Less RAM and 35% Less ROM*. accessed: 16 January, 2025. url: https://edgeimpulse.com/blog/introducing-eon.

[45] Apache TVM. *microTVM: TVM on bare-metal*. accessed: 16 January, 2025. url: https://tvm.apache.org/docs/topic/microtvm.

[46] Logan Weber and Andrew Reusch. *TinyML - How TVM is Taming Tiny*. accessed: 16 January, 2025. June 2020. url: https://tvm.apache.org/2020/06/04/tinyml-how-tvm-is-taming-tiny.

[47] STMicroelectronics. *STM32Cube.AI - STMicroelectronics - STM32 AI*. accessed: 16 January, 2025. url: https://stm32ai.st.com/stm32-cube-ai.

[48] STMicroelectronics. *Microcontrollers STM32CubeAI Solution*. accessed: 16 January, 2025. url: https://www.st.com/content/ccc/resource/sales_and_marketing/presentation/product_presentation/group0/69/82/bf/ae/5a/8b/40/91/STM32CubeAI_

press _ pres / files / STM32CubeAI _ press _ pres . pdf / jcr : content / translations / en . STM32CubeAI_press_pres.pdf.
[49] Renesas Electronics Corporation. *e-AI Development Environment for Microcontrollers*. accessed: 16 January, 2025. url: https : / / www . renesas . com / en / e - ai - development-environment-microcontrollers.
[50] Ekkono. *Edge Machine Learning and Virtual Sensors - Ekkono Solutiona*. accessed: 16 January, 2025. url: https://www.ekkono.ai.
[51] Arm. *Arm NN SDK | Efficient ML for Arm CPUs, GPUs, & NPUs - Arm*. accessed: 16 January, 2025. url: https://www.arm.com/products/silicon-ip-cpu/ethos/arm-nn.
[52] Microsoft Corporation. *The Embedded Learning Library - Embedded Learning Library (ELL)*. accessed: 16 January, 2025. url: https://microsoft.github.io/ELL.
[53] STMicroelectronics. *NanoEdgeAIStudio - Automated Machine Learning (ML) tool for STM32 developers - STMicroelectronics*. accessed: 16 January, 2025. url: https://www.st.com/en/development-tools/nanoedgeaistudio.html.
[54] STMicroelectronics. *AI:NanoEdge AI Studio - stm32mcu*. accessed: 16 January, 2025. June 2024. url: https://wiki.st.com/stm32mcu/wiki/AI:NanoEdge_AI_Studio.
[55] STMicroelectronics. *STMicroelectronics breaks down barriers to edge AI adoption with free NanoEdge AI deployment - ST News*. accessed: 16 January, 2025. url: https://newsroom.st.com/media-center/press-item.html/n4592.html.
[56] uTensor. *microTensor*. accessed: 16 January, 2025. url: https://utensor.github.io/website.
[57] Zach Shelby. *uTensor and Tensor Flow Announcement | Mbed*. accessed: 16 January, 2025. May 2019. url: https://os.mbed.com/blog/entry/uTensor-and-Tensor-Flow-Announcement.
[58] Will Lord. *Important Update on Mbed | Mbed*. accessed: 16 January, 2025. url: https://os.mbed.com/blog/entry/Important-Update-on-Mbed.
[59] Francesco Paissan. *micromind: A toolkit for tinyML research and deployment*. accessed: 16 January, 2025. url: https://github.com/micromind-toolkit/micromind.
[60] The PyTorch Foundation. *PyTorch ExecuTorch | PyTorch*. accessed: 16 January, 2025. url: https://pytorch.org/executorch-overview.
[61] Imagimob AB. *Imagimob - Edge AI | tinyML | Deep learning*. accessed: 16 January, 2025. url: https://www.imagimob.com.
[62] Arm. *OmniML Inc. - Arm*. accessed: 16 January, 2025. url: https://www.arm.com/partners/catalog/omnimlinc.
[63] Shikhar Jaiswal et al. "MinUn: Accurate ML Inference on Microcontrollers". In: *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, June 2023, pp. 26–39. isbn: 9798400701740. doi: 10.1145/3589610.3596278.
[64] Ji Lin et al. "Mcunet: Tiny deep learning on iot devices". In: *Advances in Neural Information Processing Systems* 33 (2020).
[65] Xiaying Wang et al. "FANN-on-MCU: An Open-Source Toolkit for Energy-Efficient Neural Network Inference at the Edge of the Internet of Things". In: *IEEE Internet of Things Journal* (Nov. 2019).
[66] Fraunhofer Institute for Microelectronic Circuits and Systems. *Artificial Intelligence for Embedded Systems - Fraunhofer IMS*. accessed: 16 January, 2025. url: https://www.ims.fraunhofer.de/en/Business-Unit/Industry/Industrial-AI/Artificial-Intelligence-for-Embedded-Systems-AIfES.html.
[67] Simone Salerno. *tinymlgen: Generate C code for microcontrollers from Tensorflow models*. accessed: 16 January, 2025. 2020. url: https://github.com/eloquentarduino/tinymlgen.

[68] Manuele Rusci, Alessandro Capotondi, and Luca Benini. “Memory-Driven Mixed Low Precision Quantization For Enabling Deep Network Inference On Microcontrollers”. In: *Proceedings of Machine Learning and Systems* (May 2019).
[69] Caio B. Moretti. *Neurona: Artificial Neural Networks for Arduino*. accessed: 16 January, 2025. 2016. url: https://github.com/moretticb/Neurona.
[70] Fast Machine Learning Lab. *hls4ml 1.0.0 documentation*. accessed: 16 January, 2025. url: https://fastmachinelearning.org/hls4ml/.
[71] Silicon Labs. *Silicon Labs Machine Learning Toolkit (MLTK) — MLTK 0.20.0 documentation*. accessed: 16 January, 2025. url: https://siliconlabs.github.io/mltk.
[72] Microsoft. *Neural Network Intelligence*. accessed: 16 January, 2025. Dec. 2021. url: https://github.com/microsoft/nni.
[73] Apple Inc. *Core ML | Apple Developer Documentation*. accessed: 16 January, 2025. url: https://developer.apple.com/documentation/coreml.
[74] Chengfei Lv et al. “Walle: An End-to-End, General-Purpose, and Large-Scale Production System for Device-Cloud Collaborative Machine Learning”. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, July 2022, pp. 249–265. isbn: 978-1-939133-28-1. url: https://www.usenix.org/conference/osdi22/presentation/lv.
[75] Meta. *Glow*. accessed: 16 January, 2025. url: https://ai.meta.com/tools/glow.
[76] NXP Semiconductors. *eIQ ML Software Development Environment | NXP Semiconductors*. accessed: 16 January, 2025. url: https://www.nxp.com/design/design-center/software/eiq-ml-development-environment:EIQ.
[77] Microsoft. *ONNX Runtime*. accessed: 16 January, 2025. url: https://onnxruntime.ai.
[78] Claudionor N Coelho et al. “Ultra low-latency, low-area inference accelerators using heterogeneous deep quantization with QKeras and hls4ml”. In: *arXiv preprint arXiv:2006.10159* (2020), p. 108.
[79] Plumerai. *Larq*. accessed: 16 January, 2025. url: https://larq.dev.
[80] Skymizer Taiwan Inc. *ONNC*. accessed: 16 January, 2025. url: https://onnc.ai.
[81] Latent AI. *Latent AI - Find your best model faster*. accessed: 16 January, 2025. url: https://latentai.com.
[82] Plumerai. *Plumerai*. accessed: 16 January, 2025. url: https://plumerai.com/.
[83] GreenWaves Technologies. *NNTool — GAP SDK documentation*. accessed: 16 January, 2025. url: https://greenwaves-technologies.com/manuals_gap9/gap9_sdk_doc/html/source/tools/nntool.
[84] A Burrello et al. “DORY: Automatic End-to-End Deployment of Real-World DNNs on Low-Cost IoT MCUs”. In: *IEEE Transactions on Computers* (2021), p. 1. doi: 10.1109/TC.2021.3066883.
[85] Mohammad Amin Hasanpour, Rasmus Engholm, and Xenofon Fafoutis. “Pump Cavitation Detection with Machine Learning: A Comparative Study of SVM and Deep Learning”. In: *2024 IEEE Annual Congress on Artificial Intelligence of Things (AIoT)*. 2024, pp. 219–225. doi: 10.1109/AIoT63253.2024.00050.
[86] Darius Morawiec. “sklearn-porter”. Transpile trained scikit-learn estimators to C, Java, JavaScript and others. url: https://github.com/nok/sklearn-porter.
[87] Darius Morawiec. *weka-porter: Transpile trained decision trees from Weka to C, Java or JavaScript.* accessed: 16 January, 2025. 2017. url: https://github.com/nok/weka-porter.
[88] Nikita Titov, Iaroslav Zeigerman, and Viktor Yershov. *m2cgen: Transform ML models into a native code (Java, C, Python, Go, JavaScript, Visual Basic, C#, R, Pow-*

erShell, PHP, Dart, Haskell, Ruby, F#, Rust) with zero dependencies. accessed: 16 January, 2025. url: https://github.com/BayesWitnesses/m2cgen.

[89] Simone Salerno. *micromlgen: Generate C code for microcontrollers from Python's sklearn classifiers*. accessed: 16 January, 2025. url: https://github.com/eloquentarduino/micromlgen.

[90] Lucas Tsutsui da Silva, Vinicius M. A. Souza, and Gustavo E. A. P. A. Batista. "An Open-Source Tool for Classification Models in Resource-Constrained Hardware". In: *IEEE Sensors Journal* 22 (1 Jan. 2022), pp. 544–554. issn: 1530-437X. doi: 10.1109/JSEN.2021.3128130.

[91] Qeexo AutoML. *AutoML by TDK SensEI Help Center*. accessed: 16 January, 2025. url: https://docs.qeexo.com/guides/userguides.

[92] Jon Nordby. *emlearn: Machine Learning inference engine for Microcontrollers and Embedded Devices*. Mar. 2019. doi: 10.5281/zenodo.2589394. url: https://doi.org/10.5281/zenodo.2589394.

[93] Google - The AI Edge Authors. *Build and convert models*. accessed: 3 April, 2025. url: https://ai.google.dev/edge/litert/microcontrollers/build_convert#operation_support.

[94] Tianqi Chen et al. "{TVM}: An automated {End-to-End} optimizing compiler for deep learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 578–594.

[95] Andrew Howard et al. "Searching for mobilenetv3". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2019, pp. 1314–1324.

[96] Han Cai et al. "Once-for-all: Train one network and specialize it for efficient deployment". In: *arXiv preprint arXiv:1908.09791* (2019).

[97] Matteo Gambella, Alessandro Falcetta, and Manuel Roveri. "CNAS: Constrained Neural Architecture Search". In: *2022 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE. 2022, pp. 2918–2923.

[98] Eugenio Lomurno et al. "POMONAG: Pareto-Optimal Many-Objective Neural Architecture Generator". In: *arXiv preprint arXiv:2409.20447* (2024).

[99] Xiang Liu et al. "Deep Architecture Compression with Automatic Clustering of Similar Neurons". In: *Pattern Recognition and Computer Vision: 4th Chinese Conference, PRCV 2021, Beijing, China, October 29–November 1, 2021, Proceedings, Part IV 4*. Springer. 2021, pp. 361–373.

[100] Geoffrey Hinton, Oriol Vinyals, Jeff Dean, et al. "Distilling the knowledge in a neural network". In: *arXiv preprint arXiv:1503.02531* 2.7 (2015).

[101] Nikhil P Ghanathe and Steve Wilton. "T-recx: Tiny-resource efficient convolutional neural networks with early-exit". In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. 2023, pp. 123–133.

[102] Ji Lin et al. "On-Device Training Under 256KB Memory". In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 22941–22954. url: https://proceedings.neurips.cc/paper_files/paper/2022/file/90c56c77c6df45fc8e556a096b7a2b2e-Paper-Conference.pdf.

[103] Massimo Pavan et al. "TyBox: An Automatic Design and Code Generation Toolbox for TinyML Incremental On-Device Learning". In: *ACM Transactions on Embedded Computing Systems* 23.3 (2024), pp. 1–27.

[104] Haoyu Ren, Darko Anicic, and Thomas A Runkler. "Tinyol: Tinyml with online-learning on microcontrollers". In: *2021 international joint conference on neural networks (IJCNN)*. IEEE. 2021, pp. 1–8.

[105] Andrej Karpathy. *char-rnn*. https://github.com/karpathy/char-rnn. accessed: 16 January, 2025. 2015.

[106] Google - The AI Edge Authors. *Post-training dynamic range quantization*. accessed: 16 January, 2025. url: https://ai.google.dev/edge/litert/models/post_training_quant.
[107] Google - The AI Edge Authors. *Post-training integer quantization*. accessed: 16 January, 2025. url: https://ai.google.dev/edge/litert/models/post_training_quant.
[108] Google - The AI Edge Authors. *Post-training integer quantization with int16 activations*. accessed: 16 January, 2025. url: https://ai.google.dev/edge/litert/models/post_training_integer_quant_16x8.
[109] Google - The AI Edge Authors. *Post-training float16 quantization*. accessed: 16 January, 2025. url: https://ai.google.dev/edge/litert/models/post_training_float16_quant.
[110] Youssef Abadade et al. "A comprehensive survey on tinyml". In: *IEEE Access* 11 (2023), pp. 96892–96922.
[111] Luigi Capogrosso et al. "A machine learning-oriented survey on tiny machine learning". In: *IEEE Access* 12 (2024), pp. 23406–23426.
[112] Visal Rajapakse, Ishan Karunanayake, and Nadeem Ahmed. "Intelligence at the extreme edge: A survey on reformable TinyML". In: *ACM Computing Surveys* 55.13s (2023), pp. 1–30.
[113] Soroush Heydari and Qusay H Mahmoud. "Tiny machine learning and on-device inference: A survey of applications, challenges, and future directions". In: *Sensors* 25.10 (2025), p. 3191.
[114] Emil Njor et al. "A Holistic Review of the TinyML Stack for Predictive Maintenance". In: *IEEE Access* (Dec. 2024), pp. 1–1. issn: 2169-3536. doi: 10.1109/ACCESS.2024.3512860.
[115] Mohammad Amin Hasanpour, Mikkel Kirkegaard, and Xenofon Fafoutis. "EdgeMark: An automation and benchmarking system for embedded artificial intelligence tools". In: *Journal of Systems Architecture* 167 (2025), p. 103488. issn: 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2025.103488. url: https://www.sciencedirect.com/science/article/pii/S1383762125001602.
[116] Yunhui Guo. "A survey on methods and theories of quantized neural networks". In: *arXiv preprint arXiv:1808.04752* (2018).
[117] Shaojie Zhuo et al. "An empirical study of low precision quantization for TinyML". In: *arXiv preprint arXiv:2203.05492* (2022).
[118] Pierre-Emmanuel Novac et al. "Quantization and deployment of deep neural networks on microcontrollers". In: *Sensors* 21.9 (2021), p. 2984.
[119] Markus Nagel et al. "A white paper on neural network quantization. arXiv 2021". In: *arXiv preprint arXiv:2106.08295* 4 (2021).
[120] Bryan Clark. *What is quantization aware training (qat)?* accessed: 10 July, 2025. url: https://www.ibm.com/think/topics/quantization-aware-training.
[121] Advanced Micro Devices. *Mixed Precision*. accessed: 10 July, 2025. url: https://quark.docs.amd.com/latest/onnx/tutorial_mix_precision.html.
[122] TensorFlow Contributors. *GitHub - TensorFlow Lite*. accessed: 24 July, 2025. url: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite.
[123] PyTorch Contributors. *GitHub - pytorch/QNNPACK*. accessed: 24 July, 2025. url: https://github.com/pytorch/QNNPACK.
[124] PyTorch contributors. *GitHub - pytorch quantization*. accessed: 24 July, 2025. url: https://github.com/pytorch/pytorch/tree/v2.7.1/torch/ao/quantization.
[125] Google Contributors. *GitHub - google/QKeras*. accessed: 24 July, 2025. url: https://github.com/google/qkeras.

[126] Larq Contributors. *GitHub - larq/larq*. accessed: 24 July, 2025. url: https://github.com/larq/larq.
[127] Xilinx Contributors. *GitHub - xilinx/brevitas*. accessed: 24 July, 2025. url: https://github.com/Xilinx/brevitas.
[128] Microsoft. *Neural Network Intelligence*. Version 2.0. Jan. 2021. url: https://github.com/microsoft/nni.
[129] ONNX Runtime developers. *ONNX Runtime*. Nov. 2018. url: https://github.com/microsoft/onnxruntime.
[130] Qualcomm Innovation Center. *GitHub - quic/aimet*. accessed: 24 July, 2025. url: https://github.com/quic/aimet.
[131] NNCF Contributors. *GitHub - openvinotoolkit/nncf*. accessed: 24 July, 2025. url: https://github.com/openvinotoolkit/nncf.
[132] Google. *LiteRT 8-bit quantization specification*. accessed: 24 July, 2025. url: https://ai.google.dev/edge/litert/models/quantization_spec.
[133] Google. *LiteRT model optimization*. accessed: 24 July, 2025. url: https://ai.google.dev/edge/litert/models/model_optimization.
[134] Marat Dukhan. *Accelerating TensorFlow Lite with XNNPACK Integration*. accessed: 24 July, 2025. url: https://blog.tensorflow.org/2020/07/accelerating-tensorflow-lite-xnnpack-integration.html.
[135] Marat Dukhan, Yiming Wu, and Hao Lu. *QNNPACK: Open source library for optimized mobile deep learning*. accessed: 24 July, 2025. url: https://engineering.fb.com/2018/10/29/ml-applications/qnnpack/.
[136] PyTorch Contributors. *Quantization*. accessed: 24 July, 2025. url: https://docs.pytorch.org/docs/stable/quantization.html.
[137] Plumerai Developers. *Larq - Getting started*. accessed: 24 July, 2025. url: https://docs.larq.dev/larq/.
[138] Tom Bannink et al. "Larq compute engine: Design, benchmark and deploy state-of-the-art binarized neural networks". In: *Proceedings of Machine Learning and Systems* 3 (2021), pp. 680–695.
[139] Plumerai Contributors. *Plumerai wins MLPerf Tiny 1.1 AI benchmark for microcontrollers again*. accessed: 24 July, 2025. 2023. url: https://blog.plumerai.com/2023/06/mlperf-tiny-1.1/.
[140] Plumerai contributors. *Plumerai Documentation Overview*. accessed: 24 July, 2025. url: https://docs.plumerai.com/2.2/.
[141] Advanced Micro Devices. *Brevitas Documentation*. accessed: 24 July, 2025. url: https://xilinx.github.io/brevitas/v0.12.0/.
[142] Microsoft. *Overview of NNI Model Quantization*. accessed: 24 July, 2025. url: https://nni.readthedocs.io/en/stable/compression/quantization.html.
[143] Qualcomm. *Quantization*. accessed: 24 July, 2025. url: https://docs.qualcomm.com/bundle/publicresource/topics/80-63442-50/quantization.html.
[144] ONNX Contributors. *Quantize ONNX models*. accessed: 24 July, 2025. url: https://onnxruntime.ai/docs/performance/model-optimizations/quantization.html.
[145] Qualcomm Innovation Center. *AIMET Documentation*. accessed: 24 July, 2025. url: https://quic.github.io/aimet-pages/releases/latest/index.html.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Technical
University of
Denmark

Richard Petersens Plads, Building 322
2800 Kgs. Lyngby
Tlf. 45 25 17 00

www.compute.dtu.dk