Performance of deep neural networks on low-power IoT devices

Performance of Deep Neural Networks on Low-Power IoT Devices

Christos Profentzas
Chalmers University of Technology
Gothenburg, Sweden
chrpro@chalmers.se

Magnus Almgren
Chalmers University of Technology
Gothenburg, Sweden
magnus.almgren@chalmers.se

Olaf Landsiedel
Kiel University & Chalmers University of Technology
Kiel, Germany
ol@informatik.uni-kiel.de

Abstract

Advances in deep learning have revolutionized machine learning by solving complex tasks such as image, speech, and text recognition. However, training and inference of deep neural networks are resource-intensive. Recently, researchers made efforts to bring inference to IoT edge and sensor devices which have become the prime data sources nowadays. However, running deep neural networks on low-power IoT devices is challenging due to their resource-constraints in memory, compute power, and energy. This paper presents a benchmark to grasp these trade-offs by evaluating three representative deep learning frameworks: uTensor, TF-Lite-Micro, and CMSIS-NN. Our benchmark reveals significant differences and trade-offs for each framework and its tool-chain: (1) We find that uTensor is the most straightforward framework to use, followed by TF-Micro, and then CMSIS-NN. (2) Our evaluation shows large differences in energy, RAM, and Flash footprints. For example, in terms of energy, CMSIS-NN is the most efficient, followed by TF-Micro and then uTensor, each with a significant gap.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; • **Computing methodologies** → *Neural networks.*

Keywords: IoT, Deep Neural Networks, Low-Power

ACM Reference Format:
Christos Profentzas, Magnus Almgren, and Olaf Landsiedel. 2021. Performance of Deep Neural Networks on Low-Power IoT Devices. In *Benchmarking Cyber-Physical Systems and Internet of Things (CPS-IoTBench2021), May 18, 2021, Nashville, TN, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3458473.3458823

CPS-IoTBench2021, May 18, 2021, Nashville, TN, USA

ACM ISBN 978-1-4503-8439-1/21/05…$15.00
https://doi.org/10.1145/3458473.3458823

1 Introduction

In recent years, Deep Neural Networks (DNNs) have outperformed other algorithms to solve complex problems in computer vision [10], Natural Language Processing (NLP) [5], and Human Activity Recognition (HAR) [15]. Similarly, novel IoT applications utilize DNNs to recognize and categorize sophisticated sensor data [7]. Typically, the resource-intensive tasks of training and inference are offloaded to cloud providers. With the Internet of Things, billions of devices produce massive volumes of data to be analyzed, raising two primary concerns. First, IoT devices collecting sensitive sensor data from users and storing them in cloud services poses privacy issues. Second, processing extensive amounts of sensor data may overwhelm infrastructure in terms of bandwidth and available computation. To address these concerns, researchers bring inference to edge and sensor devices [13, 16].

In particular, DNN inference on low-power IoT devices brings new challenges due to their resource-constraints. We recognize three main challenges. First, IoT devices have a small memory-size, typically in the range of KBs, where an average DNN requires MBs of storage. Second, DNN inference demands significant energy, but IoT devices require power duty cycling to preserve energy. Third, DNNs are designed using GPU/CPU optimization libraries (e.g., CUDA/Intel DL Boost), unavailable on low-power IoT devices.

The development process of DNNs on low-power IoT devices is a tedious task, where devices are manually managing memory and computation resources. There is a diversity of languages and frameworks, for example, for training DNNs in the cloud or for converting them for IoT devices, and each framework has different tool-chains. Moreover, when designing and training a DNN, it is unknown how efficient it will be on an embedded device. In this paper, we argue that it is essential to evaluate and reveal trade-offs of common frameworks, including the complexity of the development process and the resource-efficiency of their IoT run-time environments.

This paper focuses on the inference part of Deep Neural Networks (DNNs) on low-power IoT devices, assuming the training is done off-line on more powerful devices. We present a benchmark for evaluating three deep learning frameworks for IoT devices: TensorFlow-lite-Micro[2],

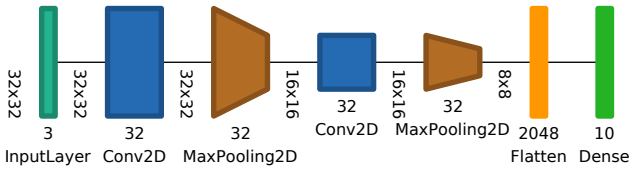


Figure 1. Overview of the Convolution Neural Network (CNN). The CNN consists of several layers stacking together: convolution kernels, max pooling, and fully connected layer. The final layer is the output number of classes.

uTensor[12], and CMSIS-NN [6]. The paper presents the following contributions:

- We design and implement a publicly available[1] benchmark to evaluate the performance of three deep learning frameworks for low-power IoT devices.
- We compare and report each framework's differences, including the development process complexity and the resource-efficiency of their run-time environment on low-power devices.
- We evaluate the resource-efficiency of prevailing DNNs on low-power IoT devices in terms of memory, computation, and energy consumption.

The rest of the paper is organized as follows. Sec. 2 provides the necessary background. Sec. 3 introduces our benchmark. Sec. 4 presents our evaluation results. Sec. 5 discusses related work and Sec. 6 concludes the paper.

2 Background and Motivation

In this section, we provide the necessary background on deep learning for low-power IoT devices.

2.1 Convolutional Neural Networks

In this paper, we focus on the widely supported Convolutional Neural Networks (CNNs). Other architectures like Recurrent Neural Networks (RNN) [3] have limited support by the three considered frameworks. Figure 1 illustrates the architecture of a CNN. The networks consist of repetitive layers of convolution kernels and pooling operations [3]. A *convolution kernel* is a matrix applying a linear transformation to an image. *Pooling operations* are down-sampling the matrix to lower dimensions by summarizing the essential features. The training process finds the values of the weights of the matrices by minimizing an objective loss function [3].

2.2 Quantization

A quantized neural network converts the weights from high precision floating points to integers. The mapping function essentially reduces the number of bits used to represent the weights. The choice of the quantization method is an open research question [4, 14].

[1] https://github.com/chrpro/TinyML-Evaluation/

Framework	Code	Generator	Dependency	Usability
uTensor	C++	Auto	Mbed OS	Easy
TF-Micro	C++	Auto	None	Medium
CMSIS-NN	C	Manual	None	Hard

Table 1. High-level comparison of deep learning frameworks. The code generator is different across the platforms. The dependency refers to other libraries needed to run inference. The usability reflects the end-to-end development process (from training to on-device inference).

2.3 IoT Deep Learning Frameworks

The frameworks used in our benchmark are the following.

uTensor. An open-source framework [12] for deep learning inference on Mbed-OS enabled IoT devices. uTensor focuses on rapid-prototyping from TensorFlow-trained neural networks to convert them for IoT devices. uTensor is written in C++ and provides built-in functions for quantization.

TensorFlow Lite Micro (TF-Micro). An open-source framework [2] for supporting machine learning inference on micro-controllers. The library is written in C++ as part of the TensorFlow ecosystem. TF-Micro uses TensorFlow to write and train a neural network. TF-Micro provides a wide range of options to optimize and quantize a neural network using the TensorFlow Lite Converter. TF-Micro can run as stand-alone or using an operating system like Zephyr OS.

CMSIS-NN. Cortex Microcontroller Software Interface Standard for Neural Networks is an open-source library [6] for ARM devices. It is written in C and provides several quantized functions like Convolution, Pooling, Softmax, and Fully-Connected layers. CMSIS-NN does not provide training tools or generators for the C code. The developer needs to use another library to train, quantize, and convert the network's weights in C code. CMSIS-NN is designed to maximize neural networks' performance on the Cortex-M series by employing the on-board DSP accelerator (CMSIS-DSP) [6].

3 Benchmark Design

In this section, we introduce the design goals of our benchmark and then present the actual benchmark.

3.1 Overview and Evaluation Goals

Our benchmark evaluates three deep-learning platforms: CMSIS-NN, uTensor, and TF-Micro (see Table 1). Their toolchains are similar in training a network but differ significantly on the code generation and quantization methods. Our goal is to evaluate trade-offs between the development process and their run-time environment's efficiency on low-power IoT devices. For the run-time environment, we focus on the efficiency of running inference on low-power devices in terms of: a) memory footprint, b) inference execution time, and c) energy consumption.

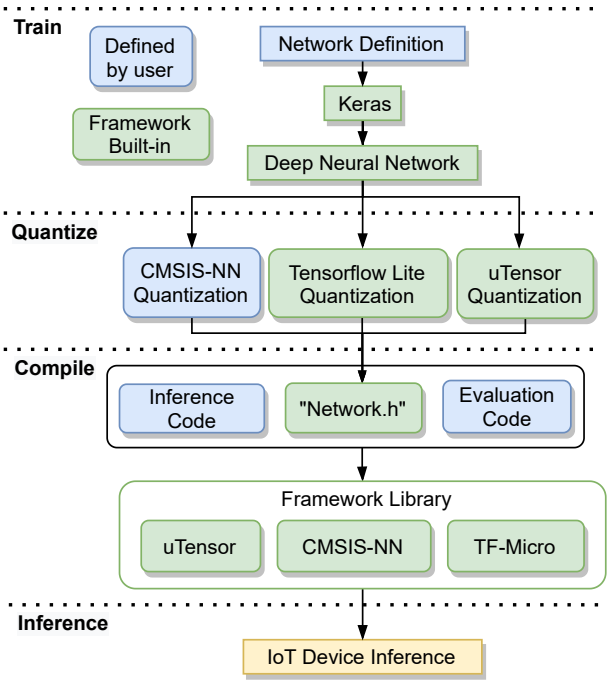


Figure 2. Benchmark's development process. The blue boxes are steps defined by the user. The green boxes are automated by the framework. We define and train all networks using Keras. We apply the quantization method provided by each framework. Finally, we compile the code for the target hardware and evaluate the inference for each framework.

Convolution - Kernels	MNIST Param.	MNIST Accuracy	CIFAR-10 Param.	CIFAR-10 Accuracy
Conv-16	6,490	0.97	8,538	0.67
Conv-32	17,578	0.98	21,674	0.68
Conv-64	53,578	0.98	61,771	0.70
Conv-96	108,010	0.99	120,298	0.71
Conv-128	180,874	0.99	197,258	0.72

Table 2. Convolutional Neural Networks trained on MNIST & CIFAR-10. The first column is the number of convolution kernels. For example, Conv-16 means the second layer (see Figure 1) and fourth layer have 16 kernels respectively. *Param.* is the number of trainable parameters.

3.2 Benchmark Process

We present the process of our benchmark in Figure 2. We color the automated steps in green and user-defined steps in blue. There are four steps to generate a neural network and execute it on an IoT device. First, we define and train the neural networks and report the accuracy in Keras[2] (see Table 2). Keras is an intuitive API wrapper on top of TensorFlow to help define and train machine learning models. All the networks are based on the same architecture (see Figure 1), data-set, and hyper-parameters. Second, we quantize the same network for each tool-chain based on each framework's method. Third, we link the network, the evaluation code, and the framework library to produce the executable file. Fourth, we run inference on the same low-power IoT device for each framework to evaluate its performance.

4 Performance Evaluation

This section presents our evaluation results on low-power IoT devices. The evaluation answers the following questions: (a) To what extent is inference technically feasible on IoT devices? (b) What is the overhead of neural networks on IoT devices in terms of computation, memory, and energy consumption? (c) What is the trade-off between automation in the development process and performance of inference on IoT devices?

4.1 Experimental Setup

Software implementation. We define and train the neural networks listed in Table 2 using Python 3.8.6 and Keras 2.4.0. For generating the object code, we have three cases:

- **CMSIS-NN.** We use the ARM GCC toolchain to build and link the C code.
- **uTensor.** We use utensor-cgen to generate the C++ code and mbed-cli to build and link C++ code, including Mbed OS.
- **TF micro.** We use the TF-Lite-Converter to export the network. We use West toolkit from Zephyr OS to build and link the TF-Micro C++ code.

Hardware Setup. We use the nRF-52840-DK board that features a 32-bit ARM Cortex-M4 at 64 MHz supporting DSP instruction set (CMSIS-DSP), 256 KB of RAM, and 1 MB of flash memory, We use the Nordic-Semiconductors Power Profiler Kit v1.1.0[3] to measure power consumption.

Data & Code Availability. We provide the data and code of the benchmark in a public repository.[4]

4.2 Data-sets

We train the neural networks with two standard data-sets.

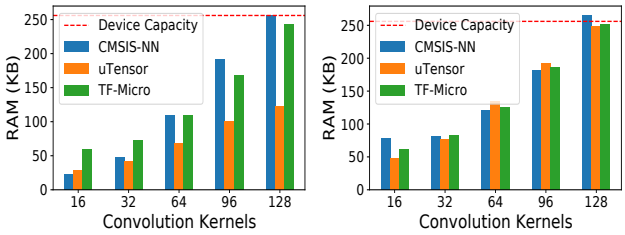
MNIST. The Modified National Institute of Standards and Technology (MNIST) data-set for hand-written numbers. The data-set consists of 70,000 black and white 28x28 pixel handwritten numbers, where 60,000 are for training and 10,000 for testing. There are ten classes, one for each digit.

CIFAR-10. The Canadian Institute For Advanced Research (CIFAR) data-set for image classification. The data-set consists of 60,000 32 x 32 color images, where 50,000 are for

[2]https://keras.io/

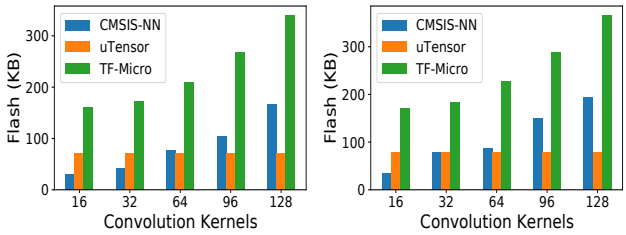
[3]https://www.nordicsemi.com/Software-and-tools/Development-Tools/Power-Profiler-Kit

[4]https://github.com/chrpro/TinyML-Evaluation/



(a) Maximum RAM usage of each CNN on MNIST, per framework. **(b)** Maximum RAM usage of each CNN on CIFAR-10, per framework.

Figure 3. RAM footprints for CMSIS-NN, uTensor, and TF-Micro. Each CNN differs by the number of the convolution kernels (see Table 2). The RAM capacity is 256 KB.



(a) Maximum flash usage of each CNN per framework on MNIST. **(b)** Maximum flash usage of each CNN per framework on CIFAR-10.

Figure 4. Flash footprints for CMSIS-NN, uTensor, and TF-Micro. Each CNN differs by the number of the convolution kernels (see Table 2). The flash capacity is 1MB.

training and 10,000 for testing. There are ten classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

4.3 Convolutional Neural Networks

For our experiment, we use the networks listed in Table 2. Each network consists of seven layers (see Figure 1), including the input and output layers. The second layer is a convolution filter with kernel size from 16 to 128, and ReLU activation function. The third layer is a max-pooling operation. The fourth layer is a convolution filter with kernel sizes from 16 to 128, and ReLU activation function. The fourth layer is a max-pooling operation. The names in Table 2 reflect the number of kernels. For example, Conv-16 means that the second and fourth layers have 16 kernels, respectively. The motivation is to scale the convolution kernels, which have most of the trainable parameters [3].

4.4 Benchmark Evaluation

We repeat each experiment 20 times. We report the maximum values of memory allocation for RAM and flash; we refer to them as footprints. We report the average inference execution time based on the cycle clock register. We report the energy consumption based on the average electric current drawn in mA, by applying 3.3 V, and report the standard deviation as ±, when it is significant.

Memory Footprints. In Figures 3, we present the RAM consumption by reporting the maximum memory allocation (footprints), for each network. In Figures 3a, we notice that for a small network with 16 kernels trained on MNIST, CMSIS-NN has a small memory footprint (22 KB) similar to uTensor and TF-Micro (28 KB & 60 KB). As we increase the kernels to 128, we notice CMSIS-NN has the largest footprint (253 KB) compared to uTensor and TF-Micro (123 KB & 243 KB). The reason is that CMSIS-NN statically allocates all the necessary data in the RAM. TF-Micro statically allocates memory for the network and dynamically allocates the buffers to store intermittent-results. uTensor dynamically allocates memory both for the network and local buffers (by utilizing Mbed OS). We notice a similar behavior when using the CIFAR-10 data-set (see Figure 3b). However, a network with 128 kernels consumes almost all the available RAM, and with CMSIS-NN (265 KB), it exceeds the device capacity (256KB). However, we consider these networks only for demonstration purposes. A real-world application using these network will not leave any memory for the applications themselves.

In Figure 4, we present the flash footprints for each framework. For CMSIS-NN and TF-Micro, the flash footprint increases proportionally with the number of static variables. The reason is that the linker allocates static variables into the flash to be copied to RAM during the start-up phase. For uTensor, the flash footprint is the same for all experiments as it counts only for program code and input vectors. Finally, we notice that the larger neural network (with 128 kernels) consumes less than 37% (367 KB) of available flash (1MB).

Inference Execution Time. We continue with the average inference execution time for each network. In Table 3, we report the results for the MNIST data-set. We notice that CMSIS-NN outperforms all other frameworks, as it is specialized for the on-board DSP accelerator available on Cortex-M devices. For example, the smallest network with 16 kernels and CMSIS-NN runs on average in 0.2 s (see Tables 2), with TF-Micro in 1.0 s, and with uTensor in 3.0 s. The largest network with 128 kernels and CMSIS-NN runs on average in 8.45 s, with TF-Micro in 43.4 s, and with uTensor in 139 s. In Table 4 we report the results for CIFAR-10 data-set. We notice a similar behavior with CMSISS-NN outperforming the others, following by TF-Micro, and uTensor. We have not noticed any significant time deviation among the experiments.

Energy Consumption. In this part, we present the energy consumption based on the average electric current drawn in mA reported by the Power Profiler Kit. Figure 5 presents a representative comparison of the electric current drawn during inference of a network with 16 kernels trained

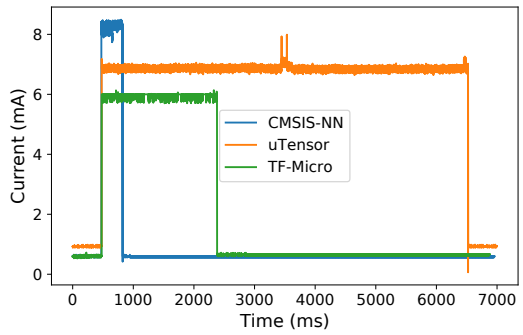


Figure 5. A representative sample of electric current drawn during inference on nRF-52840-DK, operating at 3.3 V. The neural network in this example experiment has 16 convolution kernels (see Table 2) and is trained on CIFAR-10.

on CIFAR-10 data-set. We notice a similar trend among all our experiments. CMSIS-NN has the highest current by drawing an average of 8 mA, following by uTensor with an average of 7 mA, and finally TF-Micro with an average of 6 mA. In the same figure, we see that the execution time differs strongly between the frameworks, with CMSIS-NN being the fastest due to DSP acceleration usage.

In Tables 3, we report the overall energy consumption for networks trained on MNIST data-set. The energy consumption is strongly related to the inference time and the electric current drawn during that time. We notice that CMSIS-NN has a lower energy consumption among all frameworks. For example, the smaller network with 16 kernels on CMSIS-NN consumes 5±0.1 mJ on average, followed by TF-Micro with 39±0.1 mJ on average, and uTensor with 70±0.1 mJ on average. For a large network with 128 kernels, CMSIS-NN consumes 230±0.2 mJ on average, followed by TF-Micro with 860±0.2 mJ on average, and uTensor with 3,203±0.2 on average. We notice a deviation because of the dis/charging of the PPK capacitors between the experiments. In Table 4 we report the results for CIFAR-10 data-set. We notice a similar trend with CMSISS-NN having the lowest energy consumption, following by TF-Micro and uTensor with the highest energy consumption.

4.5 Discussion and Limitations

We now discuss our results and remaining challenges.

Framework automation. We observe that defining and training a deep neural network follows a similar process among all frameworks. However, they differ significantly in development process automation and especially in the quantization method. uTensor offers an easy-to-use toolchain where built-in functions are handling the quantization and code generation. TF-Micro offers more sophisticated quantization methods, but the user needs to configure them manually. CMSIS-NN gives programmers full-control over every aspect, leading to very efficient code, but the networks need to be converted and defined manually in C code. The result is a trade-off between the complexity of the development process and the efficiency of the resulting code-base: Code in CMSIS-NN has a higher efficiency but is more complex to implement when compared to simpler and versatile approaches (TensorFlow ecosystem) with less focus on performance.

Memory Management. We observe that memory management plays a crucial role in the performance of low-power IoT devices. CMSIS-NN benefits from static allocation of memory regions and definition of specialized 8-bit and 16-bit DSP data-types. In addition, CMSIS-NN boosts performance on the Cortex-M series by employing the on-board DSP accelerator (CMSIS-DSP). On the other hand, uTensor and TF-Micro use generic 32-bit data types and dynamic memory allocation. The generic data type allows multiple platform support, but it comes with a cost on low-power IoT devices.

Concluding Remarks. We conclude the discussion with two remarks. First, there is no framework to fit both the rigorous development process of deep neural networks and the low-power devices' performance. Second, wide networks consume significant memory and energy of devices across all frameworks. Low-power IoT devices can not utilize well-established pre-trained networks available on cloud services. There is a need for ultra-lightweight neural network architectures for low-power devices.

5 Related Work

Compression. The frameworks of our evaluation supports only quantization to reduce the size of the networks. Other unsupported methods include weight pruning [4], and network compression [4].

ML Frameworks. In this paper, we use three popular frameworks for low-power devices. Next to these, other framework exist: **STM32 Cube.AI [11]** is a proprietary machine learning framework by STMicroelectronics. This framework has a limited scope and is tied to STM MCUs. **Glow [9]** is an open-source machine learning graph optimizer created by Facebook. Glow does not support low-power devices yet, but it is an on-going work.

ML Benchmarks. Our benchmark is not the first one to evaluate the performance of machine learning platforms. **MLPERF [8]** is a large-scale benchmark suite for Machine Learning inference across different platforms and hardware. **DAWNBench [1]** is a deep learning benchmark focusing mostly on GPU performance. In contrast, our paper focuses on the applicability of DNN inference on low-power IoT devices.

6 Conclusion

This paper shows the trade-offs between the development process automation of Deep Neural Networks (DNNs) and

	CMSIS-NN				uTensor				TF-Micro			
Conv. -Ker.	RAM (byte)	Flash (byte)	Time (ms)	Energy (mJ)	RAM (byte)	Flash (byte)	Time (ms)	Energy (mJ)	RAM (byte)	Flash (byte)	Time (ms)	Energy (mJ)
16	22,788	29,724	188	5±0.1	28,544	70,712	3,034	70±0.1	60,432	159,372	973	19±0.1
32	47,396	40,828	597	16±0.1	42,064	70,776	10,001	231±0.2	72,640	171,580	3,218	64±0.1
64	110,436	76,828	2,091	57±0.1	69,104	70,840	34,960	808±0.2	110,880	209,820	11,526	228±0.2
96	191,908	105,336	4,495	122±0.1	100,072	70,904	76,338	1,763±0.2	167,552	266,492	24,939	494±0.2
128	253,042	166,470	8,445	230±0.2	123,184	70,968	138,668	3,203±0.2	242,656	341,596	43,415	860±0.2

Table 3. The complete evaluation using MNIST. The RAM and flash footprints are the maximum memory allocation. Time refers to the average inference execution time. The energy consumption is based on the average electric current draw.

	CMSIS-NN				uTensor				TF-Micro			
Conv. -Ker.	RAM (byte)	Flash (byte)	Time (ms)	Energy (mJ)	RAM (byte)	Flash (byte)	Time (ms)	Energy (mJ)	RAM (byte)	Flash (byte)	Time (ms)	Energy (mJ)
16	77,720	34,076	357	10±0.1	47,704	78,248	5,986	138±0.1	62,048	170,544	1,961	39±0.1
32	80,856	78,360	1,004	27±0.1	76,504	78,264	17,148	396±0.2	82,400	184,800	5,720	113±0.1
64	120,952	87,336	2,100	58±0.1	134,104	78,280	56,824	1,313±0.2	124,736	227,136	18,641	369±0.2
96	182,288	148,660	6,021	164±0.1	191,704	78,344	119,265	2,755±0.2	185,536	287,936	38,775	768±0.2
128	**265,684**	193,068	-	-	249,304	78,472	209,073	4,830±0.2	251,736	367,136	66,113	1,309±0.2

Table 4. The complete evaluation using CIFAR-10. The RAM and flash footprints are the maximum memory allocation. Time refers to the average inference execution time. The energy consumption is based on the average electric current draw.

the low-power devices' performance. We present a benchmark to evaluate three representative frameworks for DNNs inference on low-power IoT devices. Our benchmark reveals significant differences and trade-offs for each framework and its tool-chain: (1) We find that uTensor is the easiest framework to use, followed by TF-Micro, and then CMSIS-NN. (2) Our evaluation shows large differences in energy, RAM, Flash footprints. In terms of energy, CMSIS-NN is the most efficient, followed by TF-Micro and then uTensor, each with a significant gap.

7 Acknowledgments

This work was supported by the Swedish Research Council (VR) through the project "AgreeOnIT", the Swedish Civil Contingencies Agency (MSB) through the projects "RICS2" and "RIOT", and the Vinnova-funded project "KIDSAM".

References

[1] C. Coleman, D. Narayanan, D. Kang, et al. 2017. Dawnbench: An end-to-end deep learning benchmark and competition. *Conference on Neural Information Processing Systems (NIPS).*

[2] R. David, J. Duke, A. Jain, V. J. Reddi, et al. 2020. TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. arXiv:2010.08678

[3] I. Goodfellow, Y. Bengio, and A. Courville. 2016. *Deep Learning.* MIT Press.

[4] S. Han, H. Mao, and W. J. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. arXiv:1510.00149

[5] N. Kitaev, Ł. Kaiser, and A. Levskaya. 2020. Reformer: The efficient transformer. *International Conference on Learning Representations (ICLR)* (2020).

[6] L. Lai, N. Suda, and V. Chandra. 2018. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. (2018). arXiv:1801.06601

[7] I. Mehmood, A. Ullah, K. Muhammad, et al. 2019. Efficient Image Recognition and Retrieval on IoT-Assisted Energy-Constrained Platforms From Big Data Repositories. *IEEE Internet of Things Journal.*

[8] V. J. Reddi, C. Cheng, D. Kanter, et al. 2020. MLPerf Inference Benchmark. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA).*

[9] N. Rotem, J. Fix, S. Abdulrasool, et al. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. (2018). arXiv:1805.00907

[10] A. S. Razavian, H. Azizpour, J. Sullivan, et al. 2014. CNN features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops.*

[11] STM32Cube.AI 2020. STM32Cube AI. https://www.st.com/en/ embedded -software/x-cube-ai.html.

[12] uTensor 2019. uTensor TinyML AI inference library. https://utensor.github.io/.

[13] C. Wu, D. Brooks, K. Chen, et al. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *IEEE International Symposium on High Performance Computer Architecture (HPCA).*

[14] J. Yang, X. Shen, J. Xing, et al. 2019. Quantization Networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).*

[15] J. B. Yang, M. N. Nguyen, P. P. San, et al. 2015. Deep Convolutional Neural Networks on Multichannel Time Series for Human Activity Recognition. AAAI Press.

[16] Z. Zhao, K. M. Barijough, and A. Gerstlauer. 2018. DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.*