MLOps: Open Challenges from Hardware and Software Perspective in TinyML Devices

Ahmed Mujtaba
Department of IT Convergence
Gachon University
Seongnam 13120, South Korea
ahmed709@gachon.ac.kr

Abdul Majeed & Seong Oun Hwang
Department of Computer Engineering
Gachon University
Seongnam 13120, South Korea
(ab09, sohwang)@gachon.ac.kr

Abstract

TinyML aims to enhance paradigms like healthcare, surveillance, and activity detection, etc. by scaling down Machine Learning (ML) algorithms to the level of resource-constrained devices such as microcontrollers (MCUs). MLOps practices for deploying, monitoring, and updating ML models in production, which can be challenging due to the limitations of MCU devices. Therefore, this paper highlights various key challenges for the successful training, deployment, and monitoring of ML models on MCUs and their limitations from both hardware and software perspectives. Such difficulties have an impact on the productivity, dependability, and scalability of TinyML systems.

1 Introduction

TinyML is an amalgamation of three independent fields: embedded software, embedded hardware, and machine learning algorithms, with the goal of bringing ML algorithms close to sensors on ultra-low power devices [1], such as MCUs (few milliWatts). Deep Learning (DL), Neural Networks (NNs), and other notable advancements in ML algorithms operate on considerably bigger datasets and require more resources (e.g., GPU). On the other hand, MCUs are battery-powered devices with limited resources (e.g., memory, power, and computation) and are designed to perform specific always-on functionalities. All of these constraints pose a variety of challenges to offering on-device sensor data analytics at extremely low power [2].

Previous researches [3, 1] have demonstrated the significance of hardware-software co-design in the effective creation of ML algorithms for resource-constraint platforms. However, technical issues associated with co-designing hardware and software for TinyML systems have yet to be addressed. In this paper, we explored and assessed some of the major problems associated with MLOps in the context of TinyML devices, especially from a hardware and software standpoint.

2 Challenges Towards the Hardware and Software Aspect

Figure 1 (left) depicts TinyML operations in which data is gathered and examined to extract pertinent features. Subsequently, a model is chosen and trained for deployment on the target MCU device. Thereafter, continuous monitoring of deployed models, as well as efficient handling mechanisms to update the models, must be employed. Figure 1 (right) shows the challenges in developing a full-stack TinyML system from a hardware and software viewpoint that will be covered in this section.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

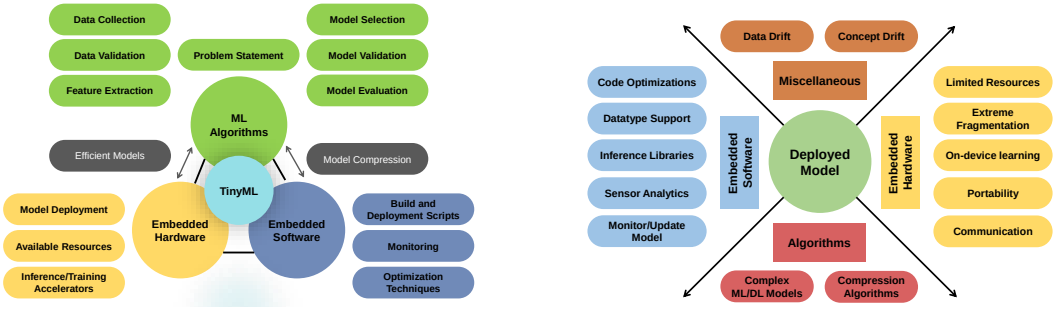

Figure 1: Tasks related to train, deploy, and update/monitor model in TinyML system (*Left*), and related challenges for efficient production of ML systems on resource-constraint platforms (*Right*).

2.1 Embedded Hardware

In this section, we discussed open challenges that are related to the TinyML devices, i.e., MCUs that are battery-operated with a minimum power budget, computation resources, and memory footprint.

2.1.1 Resource Constraints

MCUs are characterized by their minimum resources, battery-powered operation, and always-on functionality. A small memory footprint (a few KBs), constrained processor capability, and a minimal power budget (a few mWatt) are examples of minimum resources, see Appendix A. Some MCU boards contain restricted specifications and functionality, such as Floating Point Unit (FPU), Bluetooth, WiFi, and Ethernet. Depending on the application, different vendors provide different MCUs across the spectrum. Given the aforementioned limitations, monitoring/updating of ML algorithms on such devices becomes difficult, since models such as ResNets [4] or MobileNets [5–7], require a substantial amount of memory storage.

2.1.2 Fragmented MCU Ecosystem and Portability

Several businesses have recently concentrated on integrating AI into MCUs, which are frequently incorporated with sensors. There are approximately 200 billion[1] MCUs in use today, and in coming years, that number is anticipated to rise sharply. The hardware vendors provide several designs for MCUs that are customized for particular purposes in terms of power requirements, Instruction Set Architecture (ISA), a range of toolchains, and Integrated Development Environments (IDEs). The variability in design and architecture fragments the MCU ecosystem and poses a series of challenges for the deployment and monitoring of ML algorithms, which will be covered in subsequent sections.

One of the challenges is the *portability* of ML algorithms. Due to the extremely fragmented MCU ecosystem, the performance of ML models varies from device to device (e.g., inference time, optimized kernels, etc.). This challenge arised mainly due to the different design architectures, ISAs, etc. Hence, the portability of ML algorithms becomes an issue for which each device requires its own customized implementation and optimizations to the compiler level, which cannot be utilized on other hardware platforms. The fragmentation in the MCU ecosystem persists even if they are from the same manufacturer and is likely to increase in the coming years. Therefore, an early step to address this problem would significantly impact future research in TinyML for the reliable implementation of ML algorithms on MCUs. One interesting solution mentioned in [8] that utilizes containers to remove the fragmentation problem in the MCU ecosystem. However, the applicability of this solution in real production environment is not studied.

2.1.3 On-Device Upload or On-Device Learning?

ML/DL models must be timely updated when required. Therefore, monitoring ML/DL models effectively is as important as upgrading them. One of the available solutions is to enable on-device learning, which allows the model to learn on the device without having to retrain. MCUNetv3 [9]

[1] http://www.statista.com/ statistics/ 935382/worldwide-microcontroller-unit-shipments

and TinyOL [10] are the few solutions provided in the literature to enable on-device learning. These methods [9, 10] involve manual interaction with the onboard device and require the model in a C/C++ array [10] before uploading it to the MCU. However, MCUs are set up and operated in a distributed environment where the devices are both remote and inaccessible. Furthermore, on-device learning for always-on devices may quickly deplete the battery and disturb other device operations. An alternative would be to upload and download models from the device through a communication channel and fine-tune them on nearby PCs.

To upload an ML model, a communication medium is needed to have an automated interaction with the device and enable update, upload, and training when required. One approach is to use built-in communication protocols such as Serial Peripheral Interface (SPI), Universal Asynchronous Receiver/Transmitter (UART), etc. Such communication methods, however, need manual interaction with the device. Other communication protocols like WiFi, Bluetooth, etc., consume a huge amount of power [11] and appear to be poor choices for always-on battery-operated device. As a result, given the limits of the MCUs, all of the aforementioned issues necessitate an efficient method for effectively monitoring, training, and uploading the model.

2.2 ML/DL Algorithms and Embedded Software

In this section, we will discuss challenges related to the software aspect of MLOps in light of TinyML devices, including the training, compression techniques, deployment libraries, and language support for the ML and DL models.

2.2.1 More Training Parameters, More Accuracy!

In recent years, many sophisticated DL models have been proposed with the goal of increasing accuracy without considering the training parameters; see Appendix B. To train such models, a plethora of computing power and memory storage is required due to which powerful machines like GPUs being utilized. These sophisticated models cannot fit within MCUs because the required computation power and memory footprint surpass the MCU constraints. On the one hand, model compression techniques like quantization [12–29], pruning [30–35], neural architecture search [36–43], and hybrid [44, 45] algorithms—a combination of quantization, pruning, and neural architecture search—can be used to match the MCU's strict constraints with an acceptable accuracy degradation for inference. On the other hand, to support both training and inference on MCU devices, efficient lightweight models that use less memory and computation must be designed. Presently, few solutions to update models on MCUs are supported via on-device learning [10, 9], as discussed in Section 2.1.3.

2.2.2 Quantization Algorithms

Quantization is the most widely explored model compression technique for resource-constrained devices since it retains an acceptable level of accuracy with a significant reduction in model size and computations. There are two types of quantization methodologies, Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). Several baselines in PTQ including BRECQ [12], Bit-Split [13], AdaQuant [14], LAQP [15], ACIQ [16] require less time and training data than QAT including ZeroQ [17], PACT [18], DSQ [19], LSQ [20], DoReFa-Net [21], LQ-Net [22], QIL [23], APoT [24] at the cost of accuracy degradation [12], see Table in Appendix C. MQBench [46] is the recent benchmark for quantization that highlights the shortcomings of existing quantization studies which are provided in Appendix D.

2.2.3 Environmental Change

Continuous environmental change introduces an untapped dimension to the monitoring/updating of ML algorithms. ML/DL models are trained on data that the model will expect during the deployment stage. Over a period of time, the statistical properties of input data drifts away from the original distribution [47], known as data drift. The data drift causes shift in the input data distribution, resulting in poor performance of the model. On the other hand, concept drift refers to the change in the relationship between the input data and output variables [48]. When concept drift occurs, the model is not aware of the new output variables and results in missing output labels which leads to a loss of accuracy by the model. To recover the accuracy loss, the deployed model requires re-training

to correctly predict the new data distribution and missing output labels. Therefore, concept drift and data drift are the problems that a model will encounter after deployment. Hence, an efficient monitoring to detect such drifts and robust updating mechanisms must be designed for TinyML production environments. A few solutions [47, 49–51] for detecting the drifts have been proposed, however, to the best of our knowledge, no solution has been proposed for detecting drifts on TinyML devices.

2.2.4 TinyML libraries

Modern TinyML libraries like TensorFlow Lite for Microcontrollers (TFLM) [52], X-CUBE-AI [53], CMSIS [54], and CMix [55], written in C/C++, freeze the model and use it for inference only, which makes it a *static object* [10]. None of the aforementioned libraries supports training. The trained model is transformed into a C/C++ array before being uploaded onto the target MCUs [10]. Hence, the deployment and updating of models on MCUs is time-expensive and tedious process since updating a model on fresh data necessitates retraining from scratch and re-uploading it to the MCUs. Another challenge in the inference libraries for low-power devices is the *support*. For example, CMix and CMSIS inference libraries support ARM Cortex-M processors only [54, 55], which is the popular choice for ML inference [1], while X-CUBE-AI is a propriety solution [53] and supports STM32 MCUs only. TFLM provides an interpreter-based technique for inferring models on different-architecture MCUs. Interpreters are often slower than compiler-based techniques. Although the interpreter-based approach's major goal is to provide portability across the numerous MCUs available, TFLM only supports ARM Cortex-M processors, which only cover a very narrow range[2].

2.2.5 Embedded C

During inference on MCUs with popular ARM Cortex-M processors, the minimum datatype support for cutting-edge inference libraries from ARM (CMix/CMSIS) is INT8. For INT8 quantized models, the parameters (e.g., kernel weights, input image, and intermediate activations) are stored in INT8 datatypes (i.e., int8_t, uint8_t). However, for models quantized with lower precision (INT4, INT2, etc.), the parameters are packed together in the INT8 datatype. For example, four INT2 weights are packed (i.e., concatenated) together in a format of (2,2,2,2) and stored in an INT8 datatype. The parameters are packed and unpacked during inference to prevent overflow. However, the packing and unpacking operations create overhead when executed repeatedly during inference, which consumes unnecessary battery power and causes delays in the inference of ML models. Hence, for battery-powered devices, the power consumption from such redundant tasks amortizes the always-on operations and requires attention towards enabling low-datatype support for low-bit quantized models.

3 Conclusion

This study focuses on a variety of challenges in MLOps linked to TinyML devices, i.e., low-power MCUs, from both hardware and software perspective. All of these challenges reduce the scalability of TinyML, restricting its application in industries such as healthcare, agriculture, and robotics. As a result, for effective deployment, monitoring, and on-device training, the TinyML system requires rigorous hardware-software co-designed ML/DL algorithms that take into account the limitations of the low-power hardware. Little attention is paid to monitoring of deployed models, and existing solutions need manual contact with the onboard device. However, it is highly likely that if the target hardware (i.e., MCUs) is mounted in inaccessible and distant regions, this can become an ineffective solution. Therefore, various effective tools for monitoring and updating models are required for TinyML to be scalable, productive, and dependable. As a starting point in this line, we divided the challenges and problems into hardware and software perspectives and analyzed them, concretely. We hope that our study opens up a promising future research on the development of viable TinyML systems on battery-operated MCU devices.

4 Acknowledgment

This work was supported by the Institute of Information and Communications Technology Planning and Evaluation (IITP) under the High-Potential Individuals Global Training Program Grant 2021-0-

[2]https://www.tensorflow.org/lite/microcontrollers

01532 (50%) and the the ICT R&D Program Grant RS-2022-00198001 (25%), and by the National Research Foundation of Korea (NRF) under Grant 2020R1A2B5B01002145 (25%), all funded by the Korean Government through Ministry of Science and ICT (MSIT).

References

[1] Colby R Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, et al. Benchmarking tinyml systems: Challenges and direction. *arXiv preprint arXiv:2003.04821*, 2020.

[2] Muhammad Shafique, Theocharis Theocharides, Vijay Janapa Reddy, and Boris Murmann. Tinyml: Current progress, research challenges, and future roadmap. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1303–1306. IEEE, 2021.

[3] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[5] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[6] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[7] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1314–1324, 2019.

[8] Sam Leroux, Pieter Simoens, Meelis Lootus, Kartik Thakore, and Akshay Sharma. Tinymlops: Operational challenges for widespread edge ai adoption. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1003–1010. IEEE, 2022.

[9] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. On-device training under 256kb memory. *arXiv preprint arXiv:2206.15472*, 2022.

[10] Haoyu Ren, Darko Anicic, and Thomas A Runkler. Tinyol: Tinyml with online-learning on microcontrollers. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

[11] Darshana Thomas, Ross McPherson, Greig Paul, and James Irvine. Optimizing power consumption of wi-fi for iot devices: An msp430 processor and an esp-03 chip provide a power-efficient solution. *IEEE Consumer Electronics Magazine*, 5(4):92–100, 2016.

[12] Yuhang Li, Ruihao Gong, Xu Tan, Yang Yang, Peng Hu, Qi Zhang, Fengwei Yu, Wei Wang, and Shi Gu. Brecq: Pushing the limit of post-training quantization by block reconstruction. *arXiv preprint arXiv:2102.05426*, 2021.

[13] Peisong Wang, Qiang Chen, Xiangyu He, and Jian Cheng. Towards accurate post-training network quantization via bit-split and stitching. In *International Conference on Machine Learning*, pages 9847–9856. PMLR, 2020.

[14] Itay Hubara, Yury Nahshan, Yair Hanani, Ron Banner, and Daniel Soudry. Accurate post training quantization with small calibration sets. In *International Conference on Machine Learning*, pages 4466–4475. PMLR, 2021.

[15] Yury Nahshan, Brian Chmiel, Chaim Baskin, Evgenii Zheltonozhskii, Ron Banner, Alex M Bronstein, and Avi Mendelson. Loss aware post-training quantization. *Machine Learning*, 110(11):3245–3262, 2021.

[16] B Ron, J Perera, E Hoffer, and D Soudry. Aciq: Analytical clipping for integer quantization of neural networks. *arXiv preprint arXiv:1810.05723*, 2018.

[17] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Zeroq: A novel zero shot quantization framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13169–13178, 2020.

[18] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

[19] Ruihao Gong, Xianglong Liu, Shenghu Jiang, Tianxiang Li, Peng Hu, Jiazhen Lin, Fengwei Yu, and Junjie Yan. Differentiable soft quantization: Bridging full-precision and low-bit neural networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4852–4861, 2019.

[20] Steven K Esser, Jeffrey L McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S Modha. Learned step size quantization. *arXiv preprint arXiv:1902.08153*, 2019.

[21] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[22] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382, 2018.

[23] Sangil Jung, Changyong Son, Seohyung Lee, Jinwoo Son, Jae-Joon Han, Youngjun Kwak, Sung Ju Hwang, and Changkyu Choi. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4350–4359, 2019.

[24] Yuhang Li, Xin Dong, and Wei Wang. Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks. *arXiv preprint arXiv:1909.13144*, 2019.

[25] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[26] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8612–8620, 2019.

[27] Manuele Rusci, Alessandro Capotondi, and Luca Benini. Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers. *Proceedings of Machine Learning and Systems*, 2:326–335, 2020.

[28] Hamed F Langroudi, Vedant Karia, Tej Pandit, and Dhireesha Kudithipudi. Tent: efficient quantization of neural networks on the tiny edge with tapered fixed point. *arXiv preprint arXiv:2104.02233*, 2021.

[29] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[30] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European conference on computer vision (ECCV)*, pages 784–800, 2018.

[31] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime neural pruning. *Advances in neural information processing systems*, 30, 2017.

[32] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE international conference on computer vision*, pages 1389–1397, 2017.

[33] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE international conference on computer vision*, pages 2736–2744, 2017.

[34] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. *ACM SIGARCH Computer Architecture News*, 45(2):548–560, 2017.

[35] Edgar Liberis and Nicholas D Lane. Differentiable network pruning for microcontrollers. *arXiv preprint arXiv:2110.08350*, 2021.

[36] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[37] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

[38] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[39] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of Machine Learning and Systems*, 3:517–532, 2021.

[40] Edgar Liberis, Łukasz Dudziak, and Nicholas D Lane. μnas: Constrained neural architecture search for microcontrollers. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 70–79, 2021.

[41] Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul Whatmough. Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers. *Advances in Neural Information Processing Systems*, 32, 2019.

[42] Bo Lyu, Hang Yuan, Longfei Lu, and Yunye Zhang. Resource-constrained neural architecture search on edge devices. *IEEE Transactions on Network Science and Engineering*, 9(1):134–142, 2021.

[43] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. Memory-efficient patch-based inference for tiny deep learning. *Advances in Neural Information Processing Systems*, 34:2346–2358, 2021.

[44] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, Song Han, et al. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems*, 33:11711–11722, 2020.

[45] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[46] Yuhang Li, Mingzhu Shen, Jian Ma, Yan Ren, Mingxin Zhao, Qi Zhang, Ruihao Gong, Fengwei Yu, and Junjie Yan. Mqbench: Towards reproducible and deployable model quantization benchmark. *arXiv preprint arXiv:2111.03759*, 2021.

[47] Samuel Ackerman, Eitan Farchi, Orna Raz, Marcel Zalmanovici, and Parijat Dube. Detection of data drift and outliers affecting machine learning model performance over time. *arXiv preprint arXiv:2012.09258*, 2020.

[48] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46(4):1–37, 2014.

[49] Samuel Ackerman, Orna Raz, Marcel Zalmanovici, and Aviad Zlotnick. Automatically detecting data drift in machine learning classifiers. *arXiv preprint arXiv:2111.05672*, 2021.

[50] Eliran Roffe, Samuel Ackerman, Orna Raz, and Eitan Farchi. Detecting model drift using polynomial relations. *arXiv preprint arXiv:2110.12506*, 2021.

[51] Ankur Mallick, Kevin Hsieh, Behnaz Arzani, and Gauri Joshi. Matchmaker: Data drift mitigation in machine learning for large-scale systems. *Proceedings of Machine Learning and Systems*, 4:77–94, 2022.

[52] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems*, 3:800–811, 2021.

[53] STMicroelectronics. X-cube-ai, https://www.st.com/en/embedded-software/x-cube-ai.html, 2018.

[54] Liangzhen Lai, Naveen Suda, and Vikas Chandra. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601*, 2018.

[55] Alessandro Capotondi, Manuele Rusci, Marco Fariselli, and Luca Benini. Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(5):871–875, 2020.

[56] Lennart Heim, Andreas Biri, Zhongnan Qu, and Lothar Thiele. Measuring what really matters: Optimizing neural networks for tinyml. *arXiv preprint arXiv:2104.10645*, 2021.

[57] Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, and Francesco Conti. Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus. *IEEE Transactions on Computers*, 70(8):1253–1268, 2021.

[58] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE access*, 6:64270–64277, 2018.

A Appendix: Metrics and Energy Profiling of Deployed ML Models

There are two types of metrics highlighted by the [56], i.e., *proxy* metrics and *perceptible* metrics, that are used to quantify the deployed ML models. Perceptible metrics include inference latency and energy consumption of deployed model. While, proxy metrics include memory footprint, multiply-and-accumulate (MACC) operations of deployed model. However, perceptible metrics are more suitable and provide useful insights about the deployed model as compared to proxy metrics [56]. Another study [1] highlights the inference latency with optional energy consumption as a benchmark metric for TinyML systems. Other benchmarks to evaluate TinyML systems are also presented in [1]. The energy consumption of a deployed model is related to number of memory accesses which is further related to the number of MACC operations. The effect of memory accesses and number of MACC operations are also reflected in the inference latency [56]. Moreover, there is a linear relationship between inference latency and energy consumption of deployed model. This means that the energy consumption of deployed models increases as the inference latency increases. With regard to the energy profiling of ML models, a detailed energy profiling of deployed ML models is presented in [55, 57].

B Appendix: Parameters of Deep Learning Architectures

Millions of training parameters (M-params) are used in DL models to improve model accuracy, which increases the number of floating-point operations (FLOPs). The comparison in Figure 2 shows existing deep learning architectures with trainable parameters and floating-point computations.

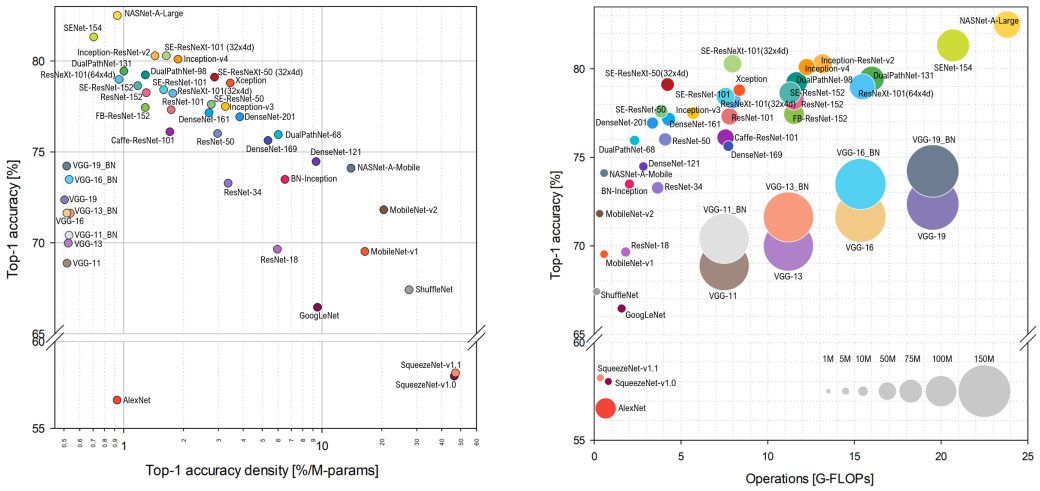

Figure 2: Top1 accuracy achieved by the DL architectures as compared to the M-params (left) and FLOPs (right), adopted from [58].

C Appendix: Comparison of QAT and PTQ Algorithms

ResNets are the most widely adopted deep neural network architecture on which several quantization algorithms have performed their experiments. Table 1 compares cutting-edge quantization algorithms for the ResNet18 and ResNet50 models.

D Appendix: Limitations of Existing Quantization Studies

MQBench [46] highlighted the two most overlooked critical factors in quantization algorithms, which are *Reproducibility* and *Deployability*. Following are some of the key insights from MQBench.

Here are some of the key insights from MQBench:

Table 1: Comparison of state-of-the-art Quantization Algorithms, (*) results are reported from the implementation of [12]. (-) are not provided by the existing study or literature.

Quantization Algorithms	**Bits (W/A)**	**Method**	**ResNet18**	**ResNet50**	**Bits (W/A)**	**ResNet18**	**ResNet50**
Full Precision	32/32		71.08	77.00	32/32	71.08	77.00
BRECQ [12]	4/4	PTQ	69.60	75.05	2/4	64.80	70.29
Bit-Split [13]	4/4	PTQ	67.56	73.71	-	-	-
AdaQuant [14]	4/4	PTQ	67.5	73.7	2/4	0.21*	0.12*
LAPQ [14]	4/4	PTQ	60.3	70.0	2/4	0.18*	0.14*
ACIQ-Mix [16]	4/4	PTQ	67.0	73.8	-	-	-
ZeroQ [17]	4/4	QAT	21.20*	2.94*	2/4	0.08*	0.08*
PACT [18]	4/4	QAT	69.2	76.5	2/2	64.4	72.2
DSQ [19]	4/4	QAT	69.56	-	2/2	65.2	-
LSQ [20]	4/4	QAT	71.2	77.6	2/2	67.9	74.6
DoReFa-Net [21]	4/4	QAT	68.1	71.4	2/2	62.6	67.1
LQ-Net [22]	4/4	QAT	69.3	75.1	2/2	64.9	71.5
QIL [23]	4/4	QAT	70.1	-	2/2	65.7	-
APoT [24]	4/4	QAT	70.7	76.6	2/2	67.3	73.4

- In many academic articles, quantization algorithms are not tested on real-world hardware platforms. As a result, these algorithms may not be regarded as reliable.
- Existing quantization algorithms are incapable of adapting to every model architecture.
- Batch normalization folding is susceptible to quantization methods and received little attention in the literature.
- Graph implementation of NN in quantization is sensitive to the model architecture.
- Only image classification tasks are evaluated extensively by the existing quantization algorithms. Other tasks like detection, recognition, etc., are not considered.

All of the aforementioned insights must be considered while formalizing a quantization algorithm; unfortunately, such aspects receive little attention.