Executive Summary of the Thesis

Neural Architecture Search for Tiny Incremental On-Device Learning

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Marco Lacava

Advisor: Prof. Manuel Roveri

Co-advisors: Matteo Gambella, Massimo Pavan

Academic year: 2022-2023

1. Introduction

Traditional Machine Learning (ML) algorithms often rely on high computational power and memory resources to process and store the data. While these requirements are feasible in resource-rich environments, they pose substantial difficulties when applied to energy-efficient solutions with constrained resources. The field of Tiny Machine Learning (TinyML) [5] focuses on deploying energy-efficient ML algorithms on small, low-power microcontrollers units (MCUs), which are commonly found in tiny devices. One of the most promising research fields in TinyML is incremental on-device learning, which consists in the ability of a TinyML model to learn and adapt to new data directly on the device itself, without constant reliance on cloud connectivity or the need to retrain the entire model offline. Given the limited resources available in tiny devices, incremental on-device learning presents a particularly challenging subject. A state-of-the-art solution that successfully addresses this challenge is TyBox, a toolbox for the automatic design and code-generation of incremental on-device TinyML models [3].
Alongside Machine Learning, Automated Machine Learning (AutoML) has emerged as a powerful technique that streamlines the ML pipeline by enabling to automatically build ML applications without much requirement for statistical and ML knowledge. A powerful AutoML approach is Neural Architecture Search (NAS), which aims to automate the design of a neural architecture optimized on its accuracy and computational requirements for a given task and dataset. Constrained Neural Architecture Search (CNAS) [2] is a novel NAS technique that integrates TinyML inspired technological constraints, enabling the development of models that can effectively operate on devices and systems with limited computational power or energy availability. However, the models produced by CNAS predominantly target mobile devices, which possess significantly greater computational resources compared to MCUs. As CNAS was not designed to accommodate the strict technological requirements of MCUs, the technique is not suitable for deployment on tiny devices. Moreover, the static networks designed by CNAS are not suitable for incremental on-device learning.
This work aims at addressing the aforementioned problems by combining CNAS with the severe constraints of MCUs, in order to obtain a model for tiny incremental on-device learning. Our goal is to design a methodology to integrate CNAS with the TyBox toolbox.

In particular, the approach presented in this paper consists of:

- the application of compression techniques to reduce the computational load and memory demand of a CNAS designed neural network;
- the implementation of an extended version of the TyBox toolbox incorporating full-integer quantization, to design an incremental on-device learning version of the reduced network.

2. Background

2.1. Pruning and quantization

One of the key challenges in deploying ML models on tiny devices is the size of the models themselves. Traditional deep learning models tend to be large and memory-intensive, making them unsuitable for deployment on constrained devices. To address this issue, model size compression techniques have emerged as effective solutions, with pruning and quantization being at the forefront.

Pruning is a technique that aims at removing weight connections from a neural network model, without compromising its predictive capabilities. By eliminating these unnecessary connections, pruning significantly reduces the number of parameters in the model, resulting in an efficiency and size improvement.

Quantization refers to the process of reducing the precision of numerical representations used in a ML model. Most models use 32-bit floating-point numbers to represent weights and activations. Quantization involves approximating these numerical values to lower bit-widths, thus enabling smaller memory storage requirements and faster computations while maintaining acceptable performance levels.

While pruning and quantization individually offer substantial model size reductions, their combination can achieve even greater compression, allowing for highly compact models that are well-suited for deployment on devices with limited resources.

2.2. TyBox

While compression techniques such as pruning and quantization have made it feasible to deploy compressed ML models on resource-constrained devices, the training process itself poses unique obstacles. Training requires a large computational load and memory demand that often exceed the capabilities of tiny devices. Most of the solutions present in the literature assume that devices only support the inference of ML models, while the training is carried out in the Cloud where appropriate computing and memory resources are available. This approach prevents embedded systems and IoT units to incrementally learn on the field as soon as new data arrive to improve their accuracy over time or to adapt to evolving working conditions.

The TyBox toolbox addresses this challenge by providing a solution for the design and code-generation of incremental on-device TinyML models [3]. TyBox is designed to receive in input a standard TinyML model $y = \Phi(I)$, where I is the input and y is the output, and the technological constraint M on the on-device RAM memory that must be satisfied by the designed incremental TinyML solution (on both inference and training phases). $\Omega(\bullet)$, the incremental version of $\Phi(\bullet)$, is composed by a fixed feature extraction block $\Phi_f(\bullet)$, an incrementally learnable classification block $\Phi_c(\bullet)$ and a buffer B. The buffer stores supervised samples coming from the field. Its primary function is to mitigate the "catastrophic forgetting" effect by retraining the incremental model on all the samples in B every time a new supervised sample becomes available. The buffer is sized to maximize the amount of stored data while respecting the constraint imposed by M. $\Phi(\bullet)$ is partitioned in two components: the feature extractor $\Phi_f(\bullet)$ and the classifier $\Phi_c(\bullet)$. This division allows for a reduction in both memory usage and computational demands, as only the classifier $\Phi_c(\bullet)$ is retrained on-device. $\Phi_f(\bullet)$ and $\Phi_c(\bullet)$ are assumed to be separated by a Flatten layer, which is the point of the model where $\Phi_f(\bullet)$ is split. Moreover, B only stores ψ_I, the features extracted by $\Phi_f(\bullet)$, and not the original input data I, hence reducing also the memory demand of the buffer. Once $\Omega(\bullet)$ has been designed, C++ codes and library implementing the inference of $\Phi_c \circ \Phi_f$ and the incremental on-device training of $\Phi_c(\bullet)$ are automatically generated. Consistency in notation has been maintained throughout this work to reference the same components.

2.3. Constrained NAS

NAS is the process of discovering the best architecture for a neural network for a specific need, automating the design of the neural network.
NAS algorithms are categorized according to three components: Search Space, which defines the set of possible architectures to be considered, Search Strategy, which establishes how to explore the search space, and Performance Estimation Strategy, which refers to the process of estimating the performance to be optimized.
NAS methods explore a search space of possible architectures and find the best one by optimizing certain objective functions. The search process involves training and evaluating multiple candidate architectures, updating the search strategy based on the performance of these architectures, and repeating the process until a satisfactory architecture is found. Finally, the optimal architecture performance is evaluated on the validation set, and some methods to speed up the performance estimation are usually applied.
A prominent family of NAS solutions is the so-called One-Shot NAS, that relies upon a Once-For-All (OFA) supernet, a network that encapsulates many possible architecture configurations. The key advantage of OFA is the decoupling of the search and training phases: instead of training each architecture individually, only the supernet is trained once on a big dataset like Imagenet and the networks are evaluated simply by inference on a different dataset, without the requirement of additional training.
Typically, NAS solutions based on OFAs aim at optimizing only the classification accuracy of the designed neural networks. Differently, Hardware-Aware NAS solutions take into account other figures of merit such as computational complexity or memory demand. An example is Evolutionary Multi Objective Surrogate-Assisted NAS (MSuNAS), an OFA based NAS that adopts a bi-objective genetic algorithm - NSGA-II - to simultaneously optimize the model's accuracy and an additional secondary objective, specifically the number of parameters, the number of MACs, or the latency of the model. MSuNAS is a surrogate-assisted NAS since it uses a surrogate accuracy predictor, that significantly accelerates the evaluation of each candidate architecture during the search process avoiding its training.
CNAS is a NAS solution that extends MSuNAS by imposing various types of constraints in the search process. Specifically, CNAS is well suited for TinyML applications because it accommodates constraints related to the technological requirements - memory and CPU clock - of the device considered for the deployment. These technological constraints are imposed in the secondary objective of the optimization that accounts jointly for the number of parameters, MACs and activations of a designed architecture.
The CNAS constrained optimization problem can be formulated as follows:

$$\begin{aligned} \text{minimize } \mathcal{G}\left(\mathcal{S}_f(\tilde{x}),\ \Phi_{CNAS}(\tilde{x})\right) \\ s.t.\ \tilde{x} \in \Omega_{\tilde{x}} \end{aligned} \tag{1}$$

where $\mathcal{G}$ is the bi-objective optimization function, $\tilde{x}$ and $\Omega_{\tilde{x}}$ represent a candidate neural network and the search space of the search, respectively, $\mathcal{S}_f$ is the accuracy of $\tilde{x}$ predicted by the surrogate model, and $\Phi_{CNAS}(\tilde{x})$ is the new secondary objective introduced by CNAS, which accounts for the technological constraints received in input by penalizing any architecture not meeting the limitations imposed by the target development device. At the end of the search, CNAS returns the set of the k optimal network architectures $X^o = \{x_1^o, \ldots, x_k^o\}$. These resulting networks achieve the best trade-off between accuracy and the hardware requirements.

3. Proposed solution

NAS significantly advances the automatic development of high-performing models. However, the transition from a discovered architecture to a deployable application involves manual coding tasks, which can be time-consuming and limit the reproducibility of research findings. By integrating NAS with automatic code-generation, it is possible to bridge the gap between research and deployment: after an architecture is discovered by NAS, the corresponding code for that architecture is generated automatically. This not only accelerates the deployment of novel architectures but also ensures consistency and reliability in the implementation process.
The focus of our work revolves around the integration of CNAS with TyBox, in order to introduce a methodology for the automated design and code generation of incremental on-device

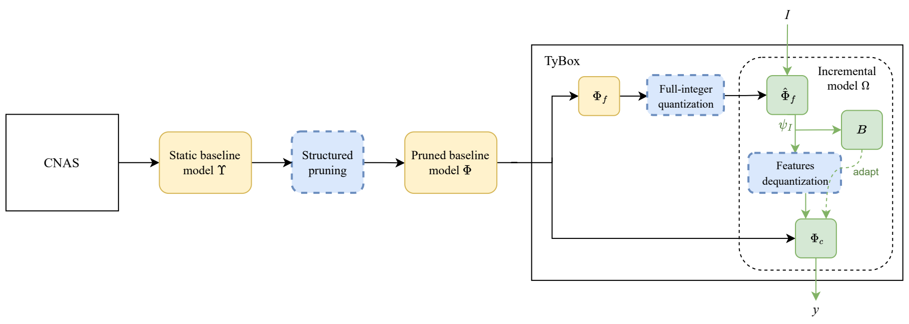

CNAS
Static baseline model Υ
Structured pruning
Pruned baseline model Φ
TyBox
Φ_f
Full-integer quantization
I
Incremental model Ω
$\hat{\Phi}_f$
ψ_I
B
Features dequantization
adapt
Φ_c
y

Figure 1: An overview of the proposed methodology. In yellow are highlighted the models and components produced in the intermediate steps, in blue the techniques applied and in green the components of the resulting incremental solution.

learning models that adhere to the strict constraints of MCUs. Specifically, our work expands the CNAS TinyML image multi-class classification case study, where the CNAS algorithm has been applied on a supernet based on MobileNetV3, a state-of-the-art Convolutional Neural Network (CNN). The architecture selected as the baseline $\Upsilon(\bullet)$ for our experiments corresponds to the smallest optimal model discovered during the CNAS search process and is described in Section 4.1. This architecture is composed of multiple non-linear convolutional blocks followed by a final Dense layer responsible for the classification task. The connection between the final layer and the preceding blocks is facilitated through a Flatten layer.
The size of $\Upsilon(\bullet)$ amounts to 8.72MB, a dimension which significantly exceeds the operational capabilities of MCUs. To achieve a reduction in model size, various compression techniques have been explored. Among the considered methods, structured pruning and post-training full-integer quantization have been selected as the chosen approaches. In contrast to conventional pruning techniques, which simply convert the weights with smaller magnitudes to zeros, structured pruning involves the effective compression of model occupation by removing its non-essential components. In our scenario, structured pruning has been applied to the Convolutional layers to eliminate their least relevant filters. The filter's relevance is estimated by computing the L1-norm of the filter's weights.
Pruning alone proves insufficient in achieving a network that meets the restrictions of tiny devices. Consequently, the TyBox toolbox has been extended to incorporate full-integer quantization. Full-integer quantization has been identified as the optimal method in this scenario due to its capability to convert model input, model output, weights, activation outputs into 8-bit integer data, compared to other quantization techniques which may leave some amount of data in floating-point. This allows to achieve up to a 4x reduction in memory usage and up to a 3x improvement in latency. Once the static baseline model $\Upsilon(\bullet)$ has been pruned, resulting in $\Phi(\bullet)$, it undergoes a decoupling process within TyBox, resulting in a feature extractor $\Phi_f(\bullet)$ and a classifier $\Phi_c(\bullet)$. $\Phi_f(\bullet)$ is composed by all the convolutional blocks of $\Phi(\bullet)$, while $\Phi_c(\bullet)$ consists solely of the final Dense layer. The primary advantage of TyBox lies in its incrementally learnable classifier, which constitutes a negligible portion of the overall model size. Hence, quantization is exclusively applied to $\Phi_f(\bullet)$, producing $\hat{\Phi}_f(\bullet)$. Since $\hat{\Phi}_f(\bullet)$ produces 8-bit outputs, while $\Phi_c(\bullet)$ is designed to receive 32-bit inputs, $\psi_I(\bullet)$ undergoes a dequantization process before being passed to $\Phi_c(\bullet)$.
A visual representation of the described methodology is given in Figure 1. The techniques implemented have been selected for their ability to considerably reduce the model size while mitigating the drop in accuracy. A detailed account of their effects is presented in Section 4.2. In addition to reducing the memory occupation of the model, the applied compression techniques contribute to a decrease in the memory demand of the TyBox buffer B. The samples stored in B correspond to the features extracted from an image by $\hat{\Phi}_f(\bullet)$. By pruning the filters of the last convolutional layer of $\Phi_f(\bullet)$ and by implementing full-integer quantization, the output ex-

tracted from $\hat{\Phi}_f(\bullet)$ is compressed, thus reducing the dimension of each latent representation. During the incremental training, the quantized buffer samples undergo dequantization in order to match the resolution of $\Phi_c(\bullet)$.

4. Experimental results

4.1. Baseline model conversion

The CNAS architecture used as the baseline Υ for the following experiments has a size of 8.72MB, and is characterized by the following configuration: *params*: 2.14M, *macs*: 6.85M, *activations*: 0.23M, *top1 accuracy*: 89.9%. The CNAS algorithm has been developed in PyTorch, while the TyBox toolbox has been designed to receive in input a model in TensorFlow (TF) file format. Given the different formats of the two solutions and the non-linearity of the CNAS model, we opted to manually convert the CNAS architecture and train it from scratch in TF. To address the conversion process, a different input image size has been adopted. The CNAS supernet was initially trained on Imagenet, which contains 256x256 color images, while the subnetworks were fine-tuned on CIFAR-10, comprising 32x32 color images that have been resized to match the resolution explored by the NAS. The PyTorch network chosen as baseline for our experiment has an input size of 40x40. Due to resource limitations, conducting the supernet training on the Imagenet dataset in TF was not feasible. Consequently, only the training on CIFAR-10 has been performed, using an input image size of 32x32. The network has been trained for 150 epochs using a custom configuration, with Adam optimizer, a learning rate of 0.0015, and early stopping (patience of 20 epochs), resulting in an accuracy of 79.29%. The poorer performance of the TF training has been attributed to the differences in training, the diverse data preprocessing between the two ML libraries and the different input image sizes required by the two networks.

4.2. Proposed methodology

This section details the results achieved by the application of the proposed methodology described in Section 3 and presented in Table 1. The target technological constraint M we imposed on the on-device RAM memory, and that must be satisfied by the final incremental TinyML solution, is 1 MB.

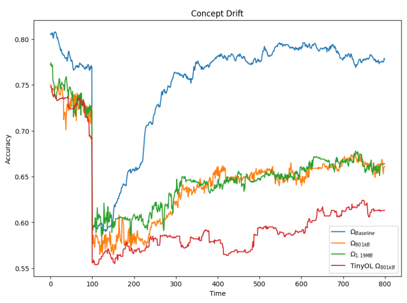
Model	Size (%)	Accuracy
Υ	8.72MB (100%)	79.29%
$\Phi_{4.31MB}$	4.31MB (49.4%)	79.48%
$\Omega_{1.19MB}$	1.19MB (13.7%)	78.50%
$\Phi_{2.89MB}$	2.89MB (33.1%)	77.15%
Ω_{801kB}	801kB (9.2%)	75.00%

Table 1: Structured pruning and full-integer quantization results. The size of each model represents the occupation of both weights and activations.

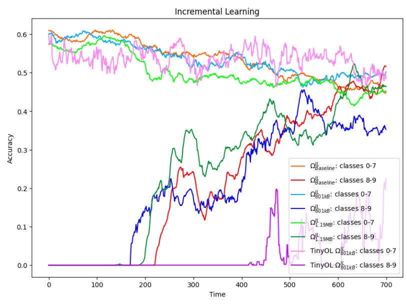
By removing the least significant convolutional filters of Υ, a compression of up to 49.4% of the original size can be achieved without exhibiting any significant accuracy drop. In accordance with the established memory constraint M, the primary model selected for deployment is $\Phi_{2.89MB}$. An evaluation of the performance of $\Phi_{4.31MB}$ has also been conducted to assess the impact of a more relaxed pruning. The implementation of full-integer quantization in TyBox induces a substantial reduction in model size, accompanied by a concurrent decrease in accuracy, that is influenced by the strictness of the previously performed pruning. Considering $\Phi_{2.89MB}$, although experiencing a 2.15% drop in accuracy, the resulting TyBox incremental model Ω_{801kB} achieves an occupation of 801kB, making it suitable for the deployment on a device with memory constraint M. The size reduction does not correspond to exactly 75% because, as anticipated in Section 3, only the FE, which constitutes the 98.8% of Ω_{801kB}, has been quantized.

4.3. Application scenarios

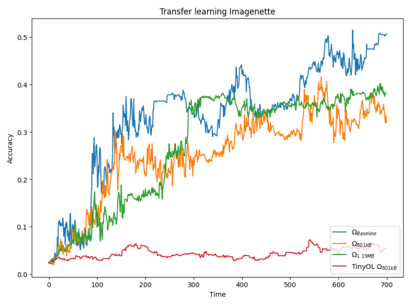
The experimental setting concerns the image classification on a multi-class problem. For this purpose, the CIFAR-10 and Imagenette [1] datasets have been considered. To validate the performance of the incremental models, we considered the same application scenarios that have been used for the validation of the TyBox multi-class case study [3]. A description of the scenarios is provided in the following subsections.
The incremental model used in this experimental setting is Ω_{801kB}, since it is the one that respects the target constraints imposed by M. Considering that during the deployment some



(a) Concept Drift.



(b) Incremental Learning.



(c) Transfer Learning.

Figure 2: Application scenarios results.

of the available 1MB memory is dedicated to the storage of libraries and code, the memory constraint imposed in the TyBox experiments is 830kB. This allows to have a buffer B of around 29kB. By applying structured pruning on the last convolutional layer of Ω_f and by implementing full-integer quantization in TyBox, each individual buffer sample requires only 187 bytes for storage, so B_{801kB} achieves a capacity of 158 samples. In each experiment, the training set is provided in an incremental manner during the experiments, while 200 elements of the test set are used to evaluate the classification accuracy after each supervised sample is provided. Two distinct solutions are considered for the comparison: the non-quantized TyBox incremental version of the original baseline model, $\Omega_{Baseline}(\bullet)$, and TinyOL [4], an alternative incremental on-device toolbox, which perform the incremental training only on the latest supervised sample received rather than all the samples stored in B, equivalent to a quantized TyBox incremental model with buffer size equal to 1. The performance of $\Omega_{1.19MB}$ are also provided, to analyze the impact of a less strict pruning. To ensure comparable results, many different learning rates have been tested for each model. Experimental results are listed in Fig. 2. The results represent the mean over 5 repetitions of the experiments. For the sake of clarity, confidence intervals have been omitted from the graphs, but have been reported in the experiments description.

4.3.1 Concept drift

With concept drift, the distribution of the data changes over time after the model has been deployed on-device. This application scenario measures the ability of the designed incremental model to recover from changes in the process generating the data. We considered an abrupt concept drift affecting the CIFAR-10 multi-class classification problem where classes 4 and 6 are swapped at sample 100. The total number of training set samples is 800. The initial adjustment in accuracy antecedent to the concept drift can be attributed to incremental learning. Although both $\Omega_{801kB}(\bullet)$ (0.658 $\pm$ 0.019) and $\Omega_{1.19MB}(\bullet)$ (0.663 $\pm$ 0.021) are able to gradually recover from the concept drift, they do not manage to reach their initial level of accuracy. This is reasonable considering the substantial compression the models have been subjected to. Notably, the TyBox $\Omega_{801kB}(\bullet)$ incremental solutions display a superior recovery compared to its TinyOL counterpart (0.613 $\pm$ 0.014). This is certainly attributed to the increased buffer size. As expected, $\Omega_{Baseline}(\bullet)$ is able to completely recover from the effects of concept drift (0.778 $\pm$ 0.007).

4.3.2 Incremental learning

With incremental learning, the task to be solved is extended after the model has been deployed on-device. This application scenario measures the ability of the model to learn a new task without forgetting the old one. More specifically, the classification problem on the CIFAR-10 dataset is initially configured comprising only classes from 0 to 7, while at sample 100, even classes 8 and 9 are included in the classification problem. 700 samples are used as incremental training set, while the test set comprises 200 samples containing all the 10 classes.

The models used in this experiment are different from those of the other two scenarios. The baseline CNAS architecture has been only trained over the classes 0-7, achieving an accuracy of 61.42%. It is noteworthy that networks that underwent a more substantial training are able to achieve higher accuracies on classes 0-7, but exhibites challenges when attempting to learn the classes 8-9. The incremental models used for the experiment have been derived from this particular baseline model $\Upsilon^{IL}(\bullet)$. Considering the distinctions in the training dataset, the largest compression achievable without introducing any significant drop in accuracy provides an incremental model $\Omega^{IL}_{954kB}(\bullet)$ of size 954kB. All the models maintain acceptable performance in the previously learned classes. $\Omega^{IL}_{954kB}(\bullet)$ is able to partially learn the new classes (0.353 ± 0.047) but struggles to reach a level of accuracy close to the old classes (0.502 ± 0.008). On the other hand, both $\Omega^{IL}_{1.19MB}(\bullet)$, benefitting from a more relaxed pruning, and $\Omega^{IL}_{Baseline}(\bullet)$, manage to achieve classification accuracies on the new classes (0.464 ± 0.047, 0.518 ± 0.089) comparable to those of the old classes (0.452 ± 0.022, 0.422 ± 0.021). In contrast, the TinyOL solution is only able to partially learn the new classes (0.243 ± 0.240) due to its limited buffer size.

4.3.3 Transfer learning

With Transfer Learning the static CNAS model is initially trained to solve a given task and, once compressed and deployed, it is incrementally re-trained on-device to address a different task. This application scenario measures the ability of the designed algorithm to transfer previously acquired knowledge to a different classification task. To test the incremental learning ability of the proposed solution, we trained $\Upsilon(\bullet)$ on CIFAR-10 and we applied the incremental learning procedure on Imagenette. This dataset has been divided into training (700 samples) and testing (200 samples). $\Omega_{Baseline}(\bullet)$ (0.505 ± 0.092), as well as $\Omega_{801kB}(\bullet)$ (0.320 ± 0.041) and $\Omega_{1.19MB}$ (0.381 ± 0.047), is able to learn the Imagenette dataset over time, although the two TyBox incremental models do not manage to attain results comparable to those achieved by $\Omega_{Baseline}(\bullet)$. This degradation in performance is undeniably related to the severity of the compression executed. Lastly, given their constrained buffer size, the TinyOL solution fail to effectively learn the new dataset (0.021 ± 0.020).

5. Conclusions

The aim of this work was to design a methodology to integrate CNAS with tiny incremental on-device learning making use of the TyBox toolbox, in order to provide an incremental version of a CNAS subnetwork that conforms with the restrictions of tiny devices. This was achieved by applying structured pruning and by extending TyBox with full-integer quantization. The effectiveness and efficiency of the proposed methodology were evaluated across diverse application scenarios. Comparative analyses were conducted against the original CNAS model and an alternate incremental on-device toolbox. Measurements showed that solutions designed with our methodology achieve performance comparable to those of the original CNAS model, and consistently outperforms the incremental models produced by the alternative toolbox. Future directions involve addressing the accuracy drop caused by the baseline model conversion, exploring the impacts of other compression techniques, and their integration in the CNAS framework.

References

[1] Fast.ai. Imagenette. `https://github.com/fastai/imagenette`, 2022.

[2] Matteo Gambella and Manuel Roveri. Searching neural architectures with constraints, 2021.

[3] Massimo Pavan, Eugeniu Ostrovan, Armando Caltabiano, and Manuel Roveri. Tybox: An automatic design and code-generation toolbox for tinyml incremental on-device learning. *ACM Trans. Embed. Comput. Syst.*, jun 2023.

[4] 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, 2021.

[5] Pete Warden and Daniel Situnayake. *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers.* O'Reilly Media, 2019.