Title

Machine Learning for Microcontroller-Class Hardware: A Review.

Journal

IEEE Sensors Journal, 22(22)

ISSN

1530-437X

Authors

Sandha, Sandeep
Srivastava, Mani
Saha, Swapnil Sayan

Publication Date

2022-11-15

DOI

10.1109/jsen.2022.3210773

Peer reviewed

Published in final edited form as:

IEEE Sens J. 2022 November 15; 22(22): 21362–21390. doi:10.1109/jsen.2022.3210773.

Machine Learning for Microcontroller-Class Hardware: A Review

Swapnil Sayan Saha [Student Member, IEEE],

Sandeep Singh Sandha,

Mani Srivastava [Fellow, IEEE]

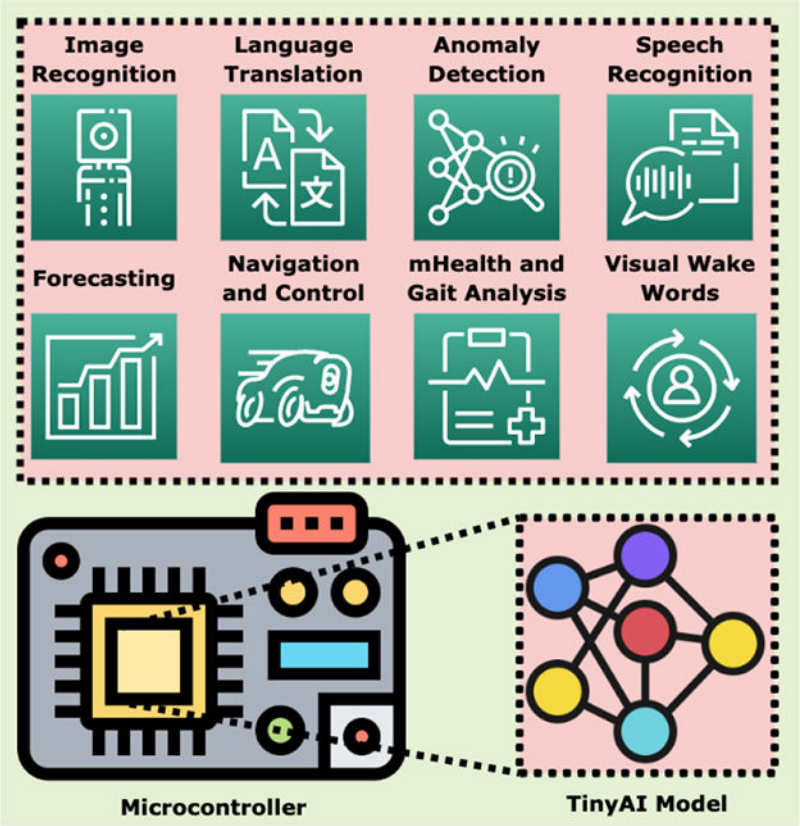
Dept. of Electrical and Computer Engineering and the Dept. of Computer Science, University of California - Los Angeles, CA 90095, USA

Abstract

The advancements in machine learning opened a new opportunity to bring intelligence to the low-end Internet-of-Things nodes such as microcontrollers. Conventional machine learning deployment has high memory and compute footprint hindering their direct deployment on ultra resource-constrained microcontrollers. This paper highlights the unique requirements of enabling onboard machine learning for microcontroller class devices. Researchers use a specialized model development workflow for resource-limited applications to ensure the compute and latency budget is within the device limits while still maintaining the desired performance. We characterize a closed-loop widely applicable workflow of machine learning model development for microcontroller class devices and show that several classes of applications adopt a specific instance of it. We present both qualitative and numerical insights into different stages of model development by showcasing several use cases. Finally, we identify the open research challenges and unsolved questions demanding careful considerations moving forward.

Graphical Abstract

swapnilsayan@g.ucla.edu .



Keywords

Feature projection; internet-of-things; machine learning; microcontrollers; model compression; neural architecture search; neural networks; optimization; sensors; TinyML

I. Introduction

Low-end Internet-of-Things (IoT) nodes such as microcontrollers are widely adopted in resource-limited applications such as wildlife monitoring, oceanic health tracking, search and rescue, activity tracking, industrial machinery debugging, onboard navigation, and aerial robotics [1] [2]. These applications limit the compute device payload capabilities, and necessitate the deployment of lightweight hardware and inference pipelines. Traditionally, microcontrollers operated on low-dimensional structured sensor data (e.g., temperature and humidity) using classical methods, making simple inferences at the edge. Recently, with the advent of machine learning, considerable endeavors are underway to bring machine learning (ML) to the edge [3] [4].

However, directly porting ML models designed for high-end edge devices such as mobile phones or single-board computers are not suitable for microcontrollers. A typical microcontroller has 128 KB RAM and 1 MB of flash, while a mobile phone can have 4 GB of RAM and 64 GB of storage [5]. The ultra resource limitations of microcontroller

class IoT nodes demand the design of a systematic workflow and tools to guide onboard deployment of ML pipelines.

This paper presents the unique requirements, challenges, and opportunities presented when developing ML models doing sensor information processing on microcontrollers. While prior surveys [3] [4] [6] [7] present a qualitative review of the model development cycle for microcontrollers, they fail to provide quantitative comparisons across alternative workflow choices and insights from application-specific case studies. In contrast, we illustrate a closed-loop workflow of ML model development and deployment for microcontroller class IoT nodes with quantitative evaluation, numerical analysis, and benchmarks showing different instances of proposed workflow across various applications. Specifically, we discuss in detail workflow components while making performance comparisons and tradeoffs of the workflow adoptions in the existing literature. Finally, we also identify bottlenecks in the current model development cycle and propose open research challenges going forward. Our contributions are as follows:

- We illustrate a coherent and closed-loop ML model development and deployment workflow for microcontrollers. We delineate each block in the workflow, providing both qualitative and numerical insights.
- We provide application-dependent quantitative evaluation and comparison of proposed workflow adaptations.
- We discuss several tradeoffs in the existing model-development process for microcontrollers and showcase opportunities and ideas in this workspace.

The rest of the paper is organized as follows: Section II outlines the TinyML workflow of model development and deployment for microcontrollers. Section III explores data engineering frameworks. Section IV shows feature projection techniques. Section V discusses model compression methods. Section VI describes lightweight ML blocks suitable for microcontrollers. Section VII discusses neural architecture search (NAS) frameworks for microcontrollers. Section VIII outlines several software suites available for porting developed models onto microcontrollers. Section IX showcases TinyML online learning frameworks. Section X provides quantitative and qualitative comparison of workflow variations depending on application. Section XI presents inter-relative and quantitative analysis of individual portions of the workflow through case studies. Section XII illustrates open challenges and ideas for future research. Section XIII provides concluding remarks.

II. TinyML Workflow

We use the term ”TinyML” to refer to model compression, machine-learning blocks, AutoML frameworks, and hardware and software suites designed to perform ultra-low-power (≤ 1 mW), always-on, and on-board sensor data analytics [4] [6] [7] on resource-constrained platforms. Typical TinyML platforms such as microcontrollers have SRAM in the order of 10^0 – 10^2 kB and flash in the order of 10^3 kB [6]. Table I provides characteristics of these devices compared to cloud servers and mobile phones. Given the widespread penetration of microcontroller-based IoT platforms in our daily lives for pervasive perception-processing-feedback applications, there is a growing push

towards embedding intelligence into these frugal smart objects [3]. Embedded AI on microcontrollers is motivated by *applicability, independence from network infrastructure, security and privacy,* and *low deployment cost:*

(i) Applicability:

Neural networks have been shown to provide rich and complex inferences over the first-principle approaches for sensor data analytics without domain expertise. With the emergence of real-time ML for microcontrollers, it is possible to turn IoT nodes from simple data harvesters or first-principles data processors to learning-enabled inference generators. TinyML combines the lightweightness of first-principle approaches with the accuracy of large neural networks.

(ii) Independence from Network Infrastructure via Remote Deployment:

Traditionally, sensor data is offloaded onto models running on mobile devices or cloud servers [19] [20]. This is not suitable for time-critical sense-compute-actuation applications such as autonomous driving [21] [22], robot control [4] [23], and industrial control system. Moreover, reliable network bandwidth or power may not be available for communicating with online models, such as in wildlife monitoring [1] or energy-harvesting intermittent systems [24] [25] [26]. TinyML allows offline and on-board inference without requiring data offloading or cloud-based inference.

(iii) Security and Privacy:

Streaming private data onto third-party cloud servers yields privacy concerns from end-users, while cybercriminals can exploit weakly protected data streams. Federated learning [27], secure aggregation [28], and homomorphic encryption [29] allow privacy-preserving and secure inference, but suffer from expensive network and compute requirement. On-board inference constrains the source and destination of private data within the IoT node itself, reducing the probability of privacy leaks and attack surfaces.

(iv) Low Deployment Cost:

While graphics processing units (GPUs) have revolutionized deep-learning [30], GPUs are energy-hungry and expensive to maintain continually for inference using small models, leading to long term financial and environmental degeneration [5]. A Cortex M4 class microcontroller costs around 5-10 USD and can run on a coin-cell battery for months, if not years [7]. TinyML allows these microcontrollers to be exploited for ultra-low-power and low-cost AI inference.

Achieving *low deployment cost* without sacrificing *performance gains* requires an unique workflow to port machine learning models onto microcontrollers compared to traditional model design. Fig. 1 illustrates the general ”closed-loop” workflow for TinyML model development and deployment. For various parts of this workflow, specific technologies and variations have emerged [6] [8] [31], which we discuss in upcoming sections. The workflow can be divided into two phases:

(i) Model Development Phase:

The phase begins by preparing a dataset from raw sensor streams using **data engineering** techniques (Section III). Data engineering frameworks are used to collect, analyze, label, and clean sensory streams to produce a dataset. Optionally, **feature projection** (Section IV) is also performed at this stage. Feature projection reduces the dimensionality of the input data through linear methods, non-linear methods, or domain-specific feature extraction. Next, several models are chosen from a pool of established **lightweight model zoo** based on the application and hardware constraints (Section VI and Section X). The zoo contains optimized blocks for well-known machine-learning primitives (e.g., convolutional neural networks, recurrent neural networks, decision trees, k-nearest neighbors, convolutional-recurrent architectures, and attention mechanisms). To achieve maximal accuracy within microcontroller SRAM, flash, and latency targets, **neural architecture search** or hyperparameter tuning is performed on candidate models from the zoo (Section VII). The hardware metrics are either obtained through proxies (approximations) or real measurements.

(ii) Model Deployment Phase:

The deployment phase begins by porting the best performing model to a **TinyML software suite** (Section VIII). These suites perform inference engine optimizations, operator optimizations, and **model compression** (Section V), along with embedded code generation. The embedded C file system is then flashed onto the microcontroller for inference. The model can be periodically fine-tuned to account for data distribution shifts using **online learning** (on-device training and federated learning) frameworks (Section IX).

To measure and compare the performance of the tinyML workflow for specific applications, Banbury *et al.* [9] proposed the widely-used MLPerf Tiny Benchmark Suite, illustrated in Table II. The benchmark contains four tasks representing a wider array of applications expected from microcontroller-class models. These include multiclass image recognition, binary image recognition, keyword spotting, and outlier detection. The benchmark suite also embraces the usage of standard datasets for each task and provides quality target metrics and model size that new workflows should aim to achieve. Hardware metrics include the working memory requirements (SRAM), model size (flash), number of multiply and add operations (MACs), and latency. From Section III to Section IX, we discuss each block in the TinyML workflow, while in Section X, we provide quantitative evaluation of the entire workflow based on applications in light of the benchmarks. In Section XI, we break down the end-to-end workflow and provide analysis of individual aspects.

III. Data Engineering

Data engineering is the practice of building systems for acquisition, analytics, and storage of data at scale [37]. Data engineering is well explored in production-scale big data systems, where robust and scalable analytics engines (e.g., Apache Spark, Apache Hadoop, Apache Hive, Apache H2O, Apache Flink, and DataBricks LakeHouse) ingest real-time sensory data via publish-subscribe paradigms (e.g., MQTT and Apache Kafka) [38]. Data streaming systems provide real-time data acquisition protocols for requirement definitions and data

gathering, while analytics engines provide support for data provenance, refinement, and sustainment. Popular general-purpose exploratory data analysis tools used in TinyML data analytics include MATLAB [39], Giotto-TDA [40], OpenCV [41], ImgAug [42], Pillow [43], Scikit-learn [44], and SciPy [45]. To suit the specific needs and goals of data engineering for TinyML systems, several specialized frameworks have emerged, illustrated in Table III.

A major challenge for enabling applications that use machine learning on microcontrollers is preparing the data and learning techniques that can automatically generalize well on unseen scenarios [33]. Thereby, most of these frameworks provide common data augmentation and data cleaning techniques such as geometric transforms, spectral transforms, oversampling, class balancing, and noise addition. MSWC [33] and Plumerai Data [36] go one step further, providing unit tests and anomaly detectors to identify problematic samples and evaluate the quality of labeled data. Plumerai Data can also automatically identify samples in the training set that are likely to be edge cases or problematic based on model performance on detected problematic samples. Such test-driven development can help users discover edge cases and outliers during model validation stages, and allow users to apply targeted augmentation, oversampling, and label correction. To reduce data collection bias, labeling errors and manual labeling effort, Edge Impulse [32], MSWC [33], SensiML DCL [34] and Plumerai Data [36] provide AI, DSP and heuristic-assisted automated labeling tools. In particular, for large-scale keyword spotting dataset generation, MSWC can automatically estimate word boundaries from audio with transcription using forced alignment and extract keywords based on user-defined heuristics in 50 languages. MSWC also automatically ensures that the generated dataset is balanced by gender and speaker diversity. Edge Impulse provides automated labeling of object detection data using YoLov5 and extraction of word boundaries from keyword spotting audio samples using DSP techniques. SensiML DCL allows video-assisted threshold-based semi-automated labeling of sensor data. Overall, these frameworks ensure that the data being used for training are relevant in context, free from bias, class-balanced, correctly labeled, contains edge cases, free from shortcuts, and encompass sufficient diversity [36].

IV. Feature Projection

An optional step in the TinyML workflow is to directly reduce the dimensionality of the data. Models operating on intrinsic dimensions of the data are computationally tractable and mitigate the curse of dimensionality. Feature projection can be divided into three types:

Linear Methods:

Linear methods for dimensionality reduction commonly used in large-scale data-mining include matrix factorization and principal component analysis (PCA) techniques such as singular value decomposition (SVD) [61], flattened convolutions [61], non-negative matrix factorization (NMF) [62], independent component analysis (ICA) [63], and linear discriminant analysis [64]. PCA is used to maximize the preservation of variance of the data in the low-dimensional manifold [65]. Among the popular linear methods, NMF is suitable for finding sparse, parts-based, and interpretable representations of non-negative data [62].

SVD is useful for finding a holistic yet deterministic representation of input data, with a hierarchical and geometric basis ordered by correlation among the most relevant variables. SVD provides a deeper factorization with lower information loss than NMF. ICA is suitable for finding independent features (blind source separation) from non-Gaussian input data [63]. ICA does not maximize variance or mutual orthogonality among the selected features. Nevertheless, linear methods are unable to model non-linearities or preserve the global relationship among features, and struggle in presence of outliers, skewed data distribution, and one-hot encoded variables.

Non-linear Methods:

Non-linear methods minimize a distance metric (e.g., fuzzy embedding topology [66], Kullback-Leibler divergence [67], local neighbourhoods [68], and Euclidean norm [69]) between the high-dimensional data and a low-dimensional latent representation. Non-linear methods to handle non-linear sampling of low-dimensional manifolds by high-dimensional vectors include locally linear embedding (LLE) [68], kernel PCA [69], t-distributed stochastic neighbor embedding (t-SNE) [70], uniform manifold approximation and projection (UMAP) [66], and autoencoders [71]. Kernel PCA couples k-NN, Dijkstra's algorithm, and partial eigenvalue decomposition to maintain geodesic distance in a low-dimensional space [69]. Similarly, LLE can be thought of as a PCA ensemble maintaining local neighborhoods in the embedding space, decomposing the latent space into several small linear functions [68]. However, both LLE and kernel PCA do not perform well with large and complex datasets. t-SNE optimizes KL-divergence between student's T distribution in the manifold-space and Gaussian joint probabilities in the higher-dimensional space [70]. t-SNE is able to reveal data structures at multiple scales, manifolds, and clusters. Unfortunately, t-SNE is computationally expensive, lacks explicit global structure preservation, and relies on random seeds. UMAP optimizes a low-dimensional fuzzy embedding to be as topologically similar as the Cech complex embedding [66]. Compared to t-SNE, UMAP provides a more accurate global structure representation, while also being faster due to the use of graph approximations. Nonetheless, while linear methods have been ported to microcontrollers [32] [72], non-linear methods are not suitable for real-time execution on microcontrollers and are usually used for visualizing high-dimensional handcrafted features.

Feature Engineering:

Feature engineering uses domain expertise to extract tractable features from the raw data [73]. Typical features include spectral and statistical features. Domain-specific feature extraction is generally more suited for micro-controllers over linear and non-linear dimensionality reduction techniques due to their relative lightweightness, as well as the availability of dedicated signal processors in commodity microcontrollers for spectral processing. However, feature engineering requires human knowledge to design statistically significant features. Feature selection can reduce the number of redundant features further during model development [74]. Feature selection methods include statistical tests, correlation modeling, information-theoretic techniques, tree ensembles, and metaheuristic methods (e.g., wrappers, filters, and embedded techniques) [75].

V. Pruning, Quantization and Encoding

Model compression aims to reduce the bitwidth and exploit the redundancy and sparsity inherent in neural networks to reduce memory and latency. Han et al. [49] first showed the concept of pruning, quantization, and Huffman coding jointly in the context of pre-trained deep neural networks (DNN). Pruning [76] refers to masking redundant weights (i.e., weights lying within a certain activation interval) and representing them in a row form. The network is then retrained to update the weights for other connections. Quantization [77] accelerates DNN inference latency by rounding off weights to reduce bit width while clustering similar ones for weight sharing. Encoding (e.g., Huffman encoding) represents common weights with fewer bits, either through conversion of dense matrices to sparse matrices [49] or smaller dense matrices through parameter redundancies [78]. Combining the three techniques can drastically reduce the size of state-of-the-art DNNs such as AlexNet (35, 6.9 MB), LeNet-5 (39, 44 kB), LeNet-300- 100 (40, 27 kB), and VGG-16 (49, 11.3 MB) without losing accuracy [49].

Common Model Compression Techniques:

Table IV showcases and compares several frameworks for model compression for microcontrollers. Among the different frameworks, TensorFlow Lite [46] is available as part of the TensorFlow training framework [79], while others are standalone libraries that can be integrated with TensorFlow or PyTorch. 88% of the frameworks provide various quantization primitives, while 50% of the frameworks support several pruning algorithms. Most of these techniques result in unstructured or random sparse patterns.

(i) Quantization Schemes: From Table IV, we can observe that the most widely-used quantization technique for microcontrollers is the fixed-precision uniform affine p**ost-training quantizer**, where a real number is mapped to a fixed-point representation via a scale factor and zero-point (offset) after training [80] [47]. Variations include quantization of weights, weights, and activations, and weights, activations, and inputs [81]. While post-training quantization (with 4, 8, and 16 bits) has been shown to reduce the model size by 4× and speed up inference by 2-3×, **quantization-aware training** is recommended for microcontroller-class models to mitigate layer-wise quantization error due to a large range of weights across channels [80] [47]. This is achieved through the injection of simulated quantization operations, weight clamping, and fusion of special layers [51], allowing up to 8× model size reduction for same or lower accuracy drop. However, care must be taken to ensure that the target hardware supports the used bitwidth. To account for distinct compute and memory requirements of different layers, **mixed-precision quantization** assigns different bit-widths for weights and activation for each layer [82]. For microcontrollers, the network subgraph is represented as a quantized convolutional layer with vectorized MAC unit, while special layers are folded into the activation function via integer channel normalization [83] [55]. Mixed-precision quantization provides 7× memory reduction [55] but is supported by limited models of microcontrollers. Recently, **binarized neural networks** [84] have been ported onto microcontroller-class hardware [52], where the weights and activations are quantized to a single bit (−1 or +1). Binarized quantization can provide 8.5-19× speedup and 8× memory reduction [53].

(ii) Pruning Algorithms: Among the different pruning algorithms, **weight pruning** is the most common, providing 4× speedup and 5-10× memory reduction [49] [48]. Weight pruning follows a schedule that specifies the type of layers to consider, the sparsity distribution to follow during training or fine-tuning, and the metric to follow when pruning (pruning policy). Common weight pruning evaluation metrics include the level and norm of weights [79] [54]. For intermittent computing systems with extremely limited power budgets, the pruning policy usually includes the energy and memory budget to maximize the collection of interesting events per unit of energy [24]. Pruning policies for intermittent computing treat pruning as a hyperparameter tuning problem, sweeping through the memory, energy, and accuracy spaces to build a Pareto frontier. Some frameworks [51] [54] provide support for structured pruning, allowing policies for channel and filter pruning rather than pruning weights in an irregular fashion.

Structured Sparsity:

Although model compression improves speedup, eliminates ineffective computations, and reduces storage and memory access costs, unstructured sparsity can induce irregular processing and waste execution time. The benefits of efficient acceleration through sparsity require special hardware and software support for storage, extraction, communication, computation, and load-balance of nonzero and trivial elements and inputs [85]. Several techniques for exploiting structured sparsity for microcontrollers have emerged. **Bayesian compression** [86] [87] assumes hierarchical, sparsity-promoting priors on channels (output activations for convolutional layers and input features for fully-connected layers) via variational inference, approximating the weight posterior by a certain distribution. For the same accuracy, Bayesian compression can reduce parameter count by 80× over unpruned models. Layer-wise **SIMD-aware weight pruning** [88] divides the weights into groups equal to the SIMD width of the microcontroller for maximal SIMD unit utilization and column index sharing. Trivial weight groups are pruned based on the root mean square of each group. SIMD-aware pruning provides 3.54× speedup and 88% reduction in model size, compared to 1.90× speedup and 80% reduction in model size provided by traditional weight pruning over unpruned models. **Differentiable network pruning** [89] performs structured channel pruning during training by applying channel-wise binary masks depending on channel salience. The size of each layer is learned through bi-level continuous gradient descent relaxation through pruning feedback and resource feedback losses without additional training overhead. Compared to traditional pruning methods, differentiable pruning provides up to 1.7× speedup, while compressing unpruned models by 80×. **Doping** [90] [91] improves the accuracy and compression factor of networks compressed using structured matrices (e.g. Kronecker products (KP)) by adding an extremely sparse matrix, using co-matrix regularization to reduce co-matrix adaptation during training. Doped KP matrices achieve a 2.5-5.5× speedup and 1.3-2.4× higher compression factor over traditional compression techniques, beating weight pruning and low-rank methods with 8% higher accuracy.

VI. Lightweight Machine Learning Blocks

To reduce the memory footprint and latency while retaining the performance of ML models running on microcontrollers, several ultra-lightweight machine learning blocks have been proposed, illustrated in Fig. 2. We describe some of these blocks in this section.

Sparse Projection:

When the input feature space is high-dimensional, sparsely projecting input features onto a low-dimensional linear manifold, called prototypes, can reduce the parameter count and improve compute efficiency of models. The projection matrix can be learned as part of the model training process using stochastic gradient descent and iterative hard thresholding to mitigate accuracy loss. Bonsai [57] is a non-linear, shallow, and sparse decision tree (DT) that can make inferences on prototypes. Similarly, ProtoNN [58] is a lightweight k-nearest neighbor (kNN) classifier that operates on prototypes.

Lightweight Spatial Convolution:

SqueezeNet [92] brought on several micro-architectural enhancements to AlexNet [93]. These include replacing 3×3 kernels with point-wise filters, decreasing input channel count using point-wise filters as a linear bottleneck, and late downsampling to enhance feature maps. The resulting network consists of stacked ”fire modules”, with each module containing a bottleneck layer (layer with point-wise filters) and an excitation layer (mix of point-wise and 3×3 filters). Using pruning, quantization, and encoding, SqueezeNet reduced the size of AlexNet by 510× (< 0.5 MB). MobileNetsV1 [12] introduced depthwise separable convolution [11] (channel-wise convolution followed by bottleneck layer), and width and resolution multipliers to control layer width and input resolution of AlexNet. Depth-wise separable convolution is 9× cheaper and induces 7-9× memory savings over 3×3 kernels. MobileNetV2 [94] introduced the concepts of inverted residuals and linear bottleneck, where a residual connection exists between bottleneck layers rather than excitation layers, and a linear output is enforced at the last convolution of a residual block. To reduce channel count, the depthwise separable convolution layer can be enclosed between the pointwise group convolution layer with channel shuffle, thereby improving the semantic relation between input and output channels across all the groups through the use of wide activation maps [95]. Instead of having residual connections across two layers, the gradient highway can act as a medium to feed each layer activation maps of all preceding layers. This is known as channel-wise feature concatenation [96] and encourages reuse and stronger propagation of low-complexity diversified feature maps and gradients while drastically reducing network parameter count.

Lightweight Multiscale Spatial Convolution:

For scalable, efficient, and real-time object detection across scales, EfficientDet [97] introduced a bidirectional feature pyramid network (FPN) to aggregate features at different resolutions with two-way information flow. The feature network topology is optimized through NAS via heuristic compound scaling of weight, depth, and resolution. EfficientDet is 4–9× smaller, uses 13-42× fewer FLOPS, and outperforms (in terms of latency and mean average precision) YOLOv3, RetinaNet, AmoebaNet, Resnet, and DeepLabV3. Scaled-

YOLOv4 [98] converts portions of FPN of YOLOv4 [99] to cross-stage partial networks [100], which saves up-to 50% computational budget over vanilla CNN backbones. Removal or fusion of batch normalization layers and downscaling input resolution can speed up multi-resolution inference by 3.6-8.8× [101] over vanilla YOLO [102] or MobileNetsV1 [12]

Low-Rank, Stabilized, and Quantized Recurrent Models:

Although recurrent neural networks (RNN) are lightweight by design, they suffer from exploding and vanishing gradient problem (EVGP) for long time-series sequences. Widely-used solutions to EVGP, namely long short-term memory (LSTM) [103], gated recurrent units [104], and unitary RNN [105] either cause loss in accuracy, or increase memory and compute overhead. FastRNN [59] solves EVGP by adding a weighted residual connection with two scalars between RNN layers to stabilize gradients during training without adding significant compute overhead. The scalars control the hidden state update extent based on inputs. FastGRNN [59] then converts the residual connection to a gate while enforcing the RNN matrices to be low-rank, sparse, and quantized. The resulting RNN is 35× smaller than gated or unitary RNN. Kronecker recurrent units [90] [106] use Kronecker products to stabilize RNN training and decompose large RNN matrices into rank-preserving smaller matrices with fewer parameters, compressing RNN by 16-50× without significant accuracy loss. Doping, co-matrix adaptation and co-matrix regularization can further compress Kronecker recurrent units by 1.3-2.4× [91]. Legendre memory units (LMU) [107], derived to orthogonalize its continuous-time history, have 10,000× more capacity while being 100× smaller than LSTM.

Temporal Convolutional Networks:

Temporal convolutional networks (TCN) [108] [109] can jointly handle spatial and temporal features hierarchically without the explosion of parameter count, memory footprint, layer count, or overfitting. TCN convolves only on current and past elements from earlier layers but not future inputs, thereby maintaining temporal ordering without requiring recurrent connections. Dilated kernels allow the network to discover semantic connections in long temporal sequences while increasing network capacity and receptive field size with fewer parameters or layers over vanilla RNN. Two TCN layers are fused through a gated residual connection for expressive non-linearity and temporal correlation modeling. A time-series TCN can be 100× smaller over a CNN-LSTM [110] [111]. TCN also supports parallel and out-of-order training.

Attention Mechanisms, Transformers, and Autoencoders:

Attention mechanisms allow neural networks to focus on and extract important features from long temporal sequences. Multi-head self-attention forms the central component in transformers, extracting domain-invariant long-term dependencies from sequences without recurrent units while being efficient and parallelizable [112]. Attention condensers are lightweight, self-contained, and standalone attention mechanisms independent of local context convolution kernels that learn condensed embeddings of the semantics of both local and cross-channel activations [113]. Each module contains an encoder-decoder architecture coupled with a self-attention mechanism. Coupled with machine design exploration,

attention condensers have been used for image classification (4.17× fewer parameters than MobileNetsV1) [114], keyword spotting (up to 507× fewer parameters over previous work) [113], and semantic segmentation (72× fewer parameters over RefineNet and Edge-SegNet) [115] at the edge. Long-short range attention (LSRA) uses two heads (convolution and attention) to capture both local and global context, expanding the bottleneck while using condensed embeddings to reduce computation cost [116]. Combined with pruning and quantization, LSRA transformers can be 18× smaller than the vanilla transformer architecture. MobileViT combines the benefits of convolutional networks and transformers by replacing local processing in convolution with global processing, allowing lightweight and low-latency transformers to be implemented using convolution [117]. Instead of using special attention and transformer blocks, transformer knowledge distillation teaches a small transformer to mimic the behavior of a larger transformer, allowing up to 7.5× smaller and 9.4× faster inference over bidirectional encoder representations from transformers [118]. Customized data layout and loop reordering of each attention kernel, coupled with quantization, has allowed porting transformers onto microcontrollers [119] by minimizing computationally intensive data marshaling operations. The use of depthwise and pointwise convolution has been shown to yield autoencoder architectures as small as 2.7 kB for anomaly detection [120].

VII. Neural Architecture Search

NAS is the automated process of finding the most optimal neural network within a neural network search space given target architecture and network architecture constraints, achieving a balance between accuracy, latency, and energy usage [125] [126] [127]. Table V compares several NAS frameworks developed for microcontrollers. There are three key elements in a hardware-aware NAS pipeline, namely the search space formulation (Section VII-A), search strategy (Section VII-B), and cost function (Section VII-C).

A. Search Space Formulation

The search space provides a set of ML operators, valid connection rules and possible parameter values for the search algorithm to explore. The neural network search space can be represented as layer-wise, cell-wise, or hierarchical [125].

Layer-wise: In layer-wise search spaces, the entire model is generated from a collection of serialized or sequential neural operators. The macro-architecture (e.g., number of layers and dimensions of each layer), initial and terminal layers of the network are fixed, while the remaining layers are optimized. The structure and connectivity among various operators are specified using variable-length strings, encoded in the action space of the search strategy [126]. Although such search spaces are expressive, layer-wise search spaces are computationally expensive, require hardcoding associations among different operators and parameters, and are not gradient friendly.

Cell-wise: For cell-wise (or template-wise) search spaces, the network is constructed by stacking repeating fixed blocks or motifs called cells. A cell is a directed acyclic graph constructed from a collection of neural operators, representing some feature transformation.

The search strategy finds the most optimal set of operators to construct the cell recursively in stages, by treating the output of past hidden layers as hidden states to apply a predefined set of ML operations on [128]. Cell-based search spaces are more time-efficient compared to layer-wise approaches and easily transferable across datasets but are less flexible for hardware specialization. In addition, the global architecture of the network is fixed.

Hierarchical: In tree-based search spaces, bigger blocks encompassing specific cells are created and optimized after cell-wise optimization. Primitive templates which are known to perform well are used to construct larger network graphs and higher-level motifs recursively, with feature maps of low-level motifs being fed into high-level motifs. Factorized hierarchical search spaces allow each layer to have different blocks without increasing the search cost while allowing for hardware specialization [129].

For applications with extreme memory and energy budget (e.g., intermittent computing systems), the search space goes down to the execution level to include operator and inference optimizations (e.g., loop transformations, data reuse, and choice of in-place operators) rather than optimizing the model at the architectural level. iNAS [124] uses RL to optimize the tile dimensions per layer, loop order in each layer, and the number of tile outputs to preserve in a power cycle for convolutional models. When combined with appropriate power-cycle energy, memory, and latency constraints, iNAS reduced intermittent inference latency by 60% compared to NAS frameworks operating at the architectural level, with a 7% increase in search overhead. Likewise, micro-TVM [130] uses a learning-enabled schedule explorer to perform automated operator and inference optimizations at the execution level. We discuss some of these optimizations in Section VIII-A, as well as operation of micro-TVM in Section VIII-B.

B. Search Strategy

The search strategy involves sampling, training and evaluating candidate models from the search space, with the goal of finding the best performing model. This is done using reinforcement learning (RL), one-shot gradient-driven NAS, evolutionary algorithms (with weight sharing), or Bayesian optimization [134]. Recent techniques, known as training-free NAS, aim to perform NAS without the costly inner-loop training [135]. Table VI compares the performance of different NAS search strategies on the ImageNet dataset for MBNetv3 [133] backbone. We distill the insights from Table VI below.

Reinforcement Learning: RL techniques, such as NASNet [128] and MNASNet [132], model NAS as a Markov Decision Process on a proxy dataset to reduce search time. RL controllers (e.g., RNNs trained via proximal policy optimization (PPo), deep deterministic policy gradient (DDPG), and Q-learning) are used to find the optimal combination of neural network cells from a pre-defined set recursively. The network graph can either consist of a series of repeatable and identical blocks (e.g., convolutional cells) whose structures are found via the controller or represented in a factorized hierarchical fashion via a layer-wise stochastic super-network. Device constraints are included in the reward function to formulate a multi-objective optimization problem. Among the different RL controllers, Q-learning-based algorithms works for simple search space (i.e, discrete and finite with tens

of parameters) [136] created through expert knowledge. PPO and DDPG are useful when the search space is complex (i.e, continuous with thousands of parameters) [137]. PPO-based on-policy algorithms are more stable than DDPG but demand more samples to converge than DDPG [138]. Overall, RL processes are slow to converge, preventing fast exploration of the search space. In addition, fine-tuning candidate networks increases search costs.

Gradient-driven NAS: Differentiable NAS using continuous gradient descent relaxation can reduce the search and training cost further on the target dataset over RL-based techniques. The goal is to learn the weights and architectural encodings through a nested bi-level optimization problem, with the gradients obtained approximately. The optimization problem can be efficiently handled using path binarization, where the weights and encodings of an over-parametrized network are alternatively frozen during gradient update using binarized gates. The final sub-network is obtained using path-level pruning. Hardware metrics are converted to a gradient-friendly continuous model before being used in the optimization function. The search space can consist of static blocks of directed acyclic graphs containing network weights, edge operations, activations, and hyperparameters or a factorized hierarchical super-network. Examples of gradient-driven NAS include DARTS [139], FBNet [140], ProxylessNAS [129], and MicroNets [8]. Drawbacks of include high GPU memory consumption and training time due to large super-network parameter count and inability to generalize across a broad spectrum of target hardware, requiring the NAS process to be repeated for new hardware.

Evolutionary Search with Weight Sharing: To eliminate the need for performing NAS for different hardware platforms separately and reduce the training time of candidate networks, several weight-sharing mechanisms (WS-NAS) have emerged [5] [31] [121] [143] [144]. WS-NAS decouples training from search by training a ”once-for-all” super-network consisting of several sub-networks which fits the constraints of eclectic target platforms. Evolutionary search is used during the search phase, where the best performing sub-networks are selected from the super-network, crossed, and mutated to produce the next generation of candidate subnetworks. Progressive shrinking and knowledge distillation ensure all the sub-networks are jointly fine-tuned without interfering with large sub-networks. Evolutionary search can also be applied to RL search spaces [145] for faster convergence or applied on several candidate architectures not part of a super-network [122]. Nevertheless, evolutionary WS-NAS suffers from excessive computation and time requirements due to super-network training, exacerbated by fine-tuning of candidate networks and slow convergence of evolutionary algorithms.

Bayesian Optimization: When training infrastructure is weak, the search space and hardware metrics are discontinuous, and the training cost per model is high, Bayesian NAS is used as a black-box optimizer. Given their problem-independent nature, Bayesian NAS can be applied across different datasets and heterogenous architectures without being constrained to one specific type of network (e.g., CNN or RNN), provided support for conditional search. The performance of the optimizer is highly dependent on the surrogate model [146]. The most widely adopted surrogate model is the Gaussian process, which allows uncertainty metrics to propagate forward while looking for a Pareto-optimal frontier

and is known to outperform other choices like random forest or Tree of Parzen Estimators [146]. The acquisition function decides the next set of parameters from the search space to sample from, balancing exploration and exploitation. The loss function is modeled as a constrained single-objective or scalarized multi-objective optimization problem. Examples include SpArSe [86], Vizier [147], and THIN-Bayes [123]. Unfortunately, Bayesian NAS does not perform well in high dimensional search spaces (e.g., performance degrades beyond a dozen parameters [148]). Moreover, Bayesian optimizers are typically used to optimize hyperparameters for fixed network architectures instead of multiple architectures as the Gaussian process does not directly support conditional search across architectures. Only THIN-Bayes can sample across different architectures thanks to support for conditional search via multiple Gaussian surrogates [123].

Training-Free NAS: Training-free NAS estimates the accuracy of a neural network either by using proxies developed from architectural heuristics of well-known network architectures [135] or by using a graph neural network (GNN) to predict the accuracy of models generated from a known search space [149] [150]. Examples of gradient-based accuracy proxies include the correlation of ReLU activations (Jacobian covariance) between minibatch datapoints at CNN initialization [151], the sum of the gradient Euclidean norms after training with a single minibatch datapoints [152], change in loss due to layer-level pruning after training with a single minibatch datapoints (Fisher) [152], change in loss due to parameter pruning after training with a single minibatch data-points (Snip) [153], change in gradient norm due to parameter pruning after training with a single minibatch datapoints (Grasp) [154], the product of all network parameters (Synaptic Flow) [155], the spectrum of the neural tangent kernel [156], and the number of linear regions in the search space [156]. Gradient-based proxies still require the use of a GPU for gradient calculation. Recently, Li *et al.* [135] proposed a gradient-free accuracy proxy, namely the sum of the average degree of each building block in a CNN from a network topology perspective. Unfortunately, both proxies and GNN suffer from the lack of generalizability across different datasets, model architectures, and design space, while the latter also suffers from the training cost of the accuracy prediction network itself.

C. Cost Function

The cost function provides numerical feedback to the search strategy about the performance of a candidate network. Common parameters in the cost function include network accuracy, SRAM usage, flash usage, latency, and energy usage. The goal of NAS is to find a candidate network that finds the extrema of the cost function, i.e., the cost function can be thought of as seeking a Pareto-optimal configuration of network parameters.

Cost Function Formulation: The cost function can be formulated as either a single or multi-objective optimization problem. Single objective optimization problems only optimize for model accuracy. To take hardware constraints into account, single-objective optimization problems are usually treated as constrained optimization problems with hardware costs acting as regularizers [123]. Multi-objective cost functions are usually transformed into a single objective optimization problem via weighted-sum or scalarization techniques [86] or solved using genetic algorithms.

Hardware Profiling: Hardware-aware NAS employs hardware-specific cost functions or search heuristics via hardware profiling. The target hardware can be profiled in real-time by running sampled models on the actual target device (hardware-in-the-loop), estimated using lookup tables, prediction models, and silicon-accurate emulators [157] or analytically estimated using architectural heuristics. Common hardware profiling techniques are shown in Table VII. Hardware-in-the-loop is slowest but most accurate during NAS runtime, while analytical estimation is fastest but least accurate [125] [134]. Examples of analytical models for microcontrollers include using FLOPS as a proxy for latency [8] [122], and standard RAM usage model [86] for working memory estimation. Recently, latency prediction models have been made more accurate through kernel (execution unit) detection and adaptive sampling [158]. For intermittent computing systems, the latency is the time required for progress preservation (writing progress indicators and computed tile outputs to flash at the end of a power cycle), progress recovery (system reboot, loading progress indicators, and tiled outputs into SRAM), battery recharge, and running inference (cost of computing multiple tiles per energy cycle) [124]. The SRAM usage in such systems is the sum of memory consumed by the input feature map, weights, and output feature map, dependent upon the tile dimensions, loop order, and preservation batch size in the search space [124].

VIII. TinyML Software Suites

After the best model is constructed from lightweight ML blocks through NAS, the model needs to be prepared for deployment onto microcontrollers. This is performed by TinyML software suites, which generate embedded code and perform operator and inference engine optimizations, some of which are shown in Fig. 3 and discussed in Section VIII-A. In addition, some of these frameworks also provide inference engines for resource management and model execution during deployment. We discuss features of notable TinyML software suites in Section VIII-B.

A. Operator and Inference Optimizations

All TinyML software suites perform several operator and inference engine optimizations to improve data locality, memory usage, and spatiotemporal execution [162]. Common techniques include the use of fused or in-place operators [130], loop transformations [161], and data reuse (output sharing or value sharing) [163].

In-Place and Fused Operators: Operator fusion or folding combines several ML operators into a specialized kernel without saving the intermediate feature representations in memory (known as in-place activation) [130]. The software suites follow user-defined rules for operator fusion depending on graph operator type (e.g., injective, reduction, complex-out fusable, and opaque) [130]. Use of fused and in-place operators have been shown to reduce memory usage by 1.6× [31] and improve speedup by 1.2-2× [130].

Loop Transformations: Loop transformations aim to improve spatiotemporal execution and inference speed by reducing loop overheads [161]. Common loop transformations include loop reordering, loop reversal, loop fusion, loop distribution, loop unrolling,

and loop tiling [161] [162] [161] [160]. Loop reordering (and reversal) finds the loop permutation that maximizes data reuse and spatiotemporal locality. Loop fusion combines different loop nests into one, thereby improving temporal locality, and increasing data locality and reuse by creating perfect loop nests from imperfect loop nests. To enable loop permutation for loop nests that are not permutable, loop distribution breaks a single loop into multiple loops [161]. Loop unrolling helps eliminate branch penalties and helps hide memory access latencies [130]. Loop tiling improves data reuse by diving the loops into blocks while considering the size of each level of memory hierarchy [160].

Data Reuse: Data reuse aims to improve data locality and reduce memory access costs. While data reuse is mostly achieved through loop transformations, several other techniques have also been proposed. CMSIS-NN provides special pooling and multiplication operations to promote data reuse [164]. TF-Net [163] proposed the use of direct buffer convolution on Cortex-M microcontrollers to reduce input unpacking overhead, which reuses inputs in the current window unpacked in a buffer space for all weight filters. Input reuse reduces SRAM usage by 2.57× and provides 2× speedup. Similarly, for GAP8 processors, the PULP-NN library provides a reusable im2col buffer (height-width-channel data layout) to reduce im2col creation overhead [165] [166], providing partial spatial data reuse. PULP-NN also features register-level data reuse, achieving 20% speedup over CMSIS-NN and 1.9× improvement over native GAP8-NN libraries.

B. Notable TinyML Software Suites

Notable open-source TinyML frameworks and inference engines include TensorFlow Lite Micro [167] [46], uTensor [168], uTVM [130], Microsoft EdgeML [57] [58] [59] [60], [169–171], CMSIS-NN [164], EloquentML [72], Sklearn Porter [174], EmbML [175], and FANN-on-MCU [176]. Closed-source TinyML frameworks and inference engines include STM32Cube.AI [172], NanoEdge AI Studio [173], Edge Impulse EON Compiler [32], TinyEngine [31] [121], Qeexo AutoML [35], Deeplite Neutrino [177], Imagimob AI [178], Neuton TinyML [179], Reality AI [180], and SensiML Analytics Studio and Knowledge Pack [34]. Table VIII compares the features of some of these frameworks.

TensorFlow Lite Micro: TensorFlow Lite Micro (TFLM) [46] [167] is a specialized version of TFLite aimed towards optimizing TF models for Cortex-M and ESP32 MCU. TFLite Micro embraces several embedded runtime design philosophies. TFLM drops uncommon features, data types, and operations for portability. It also avoids specialized libraries, operating systems, or build-system dependencies for heterogeneous hardware support and memory efficiency. TFLM avoids dynamic memory allocation to mitigate memory fragmentation. TFLM interprets the neural network graph at runtime rather than generating C++ code to support easy pathways for upgradability, multi-tenancy, multi-threading, and model replacement while sacrificing finite savings in memory. Fig 4 summarizes the operation of TFLM. TFLM consists of three primary components. First, the **operator resolver** links only essential operations to the model binary file. Second, TFLM pre-allocates a contiguous memory stack called the **arena** for initialization and storing runtime variables. TFLM uses a two-stack allocation strategy to discard initialization variables after their lifetime, thereby minimizing memory consumption. The space between

the two stacks is used for temporary allocations during memory planning, where TFLM uses bin-packing to encourage memory reuse and yield optimal compacted memory layouts during runtime. Lastly, TFLM uses an **interpreter** to resolve the network graph at runtime, allocate the arena, and perform runtime calculations. TFLM was shown to provide 2.2× speedup and 1.08× memory and flash savings over CMSIS-NN for image recognition [31].

uTensor: uTensor [168] generates C++ files from TF models for Mbed-enabled boards, aiming to generate models of < 2 kB in size. It is subdivided into two parts. The **uTensor core** provides a set of optimized runtime data structures and interfaces under computing constraints. The **uTensor library** provides default error handlers, allocators, contexts, ML operations, and tensors built on the core. Basic data types include integral type, uTensor strings, tensor shape, and quantization primitives borrowed from TFLite. Interfaces include the memory allocator interface, tensor interface, tensor maps, and operator interface. For memory allocation, uTensor uses the concept of arena borrowed from TFLM. In addition, uTensor boasts a series of optimized (built to run CMSIS-NN under the hood), legacy, and quantized ML operators consisting of activation functions, convolution operators, fully-connected layers, and pooling.

uTVM: micro-TVM [130] extends the TVM compiler stack to run models on bare-metal IoT devices without the need for operating systems, virtual memory, or advanced programming languages. micro-TVM first generates a high-level and quantized computational graph (with support for complex data structures) from the model using the **relay module**. The functional representation is then fed into the TVM **intermediate representation module**, which generates C-code by performing operator and loop optimizations via AutoTVM and Metascheduler, procedural optimizations, and graph-level modeling for whole program memory planning. AutoTVM consists of an automatic **schedule explorer** to generate promising and valid operator and inference optimization configurations for a specific microcontroller, and an XGBoost model to predict the performance of each configuration based on features of the lowered loop program. The developer can either specify the configuration parameters to explore using a schedule template specification API, or possible parameters can be extracted from the hardware computation description written in the tensor expression language. AutoTVM has lower data and exploration costs than black-box optimizers (e.g., ATLAS [181]), and provides more accurate modeling than polyhedral methods [182] without needing a hardware-dependent cost model. The generated code is integrated alongside the TVM C runtime, built, and flashed onto the device. Inference is made on the device using a graph extractor. AutoTVM was shown to generate code that is only 1.2× slower compared to handcrafted CMSIS-NN-based code for image recognition.

Microsoft EdgeML: EdgeML provides a collection of lightweight ML algorithms, operators, and tools aimed towards deployment on Class 0 devices, written in PyTorch and TF. Included algorithms include Bonsai [57], ProtoNN [58], FastRNN [59], FastGRNN [59], ShallowRNN [169], EMI-RNN [170], RNNPool [60], and DROCC [171]. EMI-RNN exploits the fact that only a small, tight portion of a time-series plot for a certain class contributes to the final classification while other portions are common among all classes. Shallow-RNN is a hierarchical RNN architecture that divides the time-series signal into

various blocks and feeds them in parallel to several RNNs with shared weights and activation maps. RNNPool is a non-linear pooling operator that can perform ”pooling” on intermediate layers of a CNN by a downsampling factor much larger than 2 (4-8×) without losing accuracy while reducing memory usage and decreasing compute. Deep robust one-class classifier (DROCC) is an OCC under limited negatives and anomaly detector without requiring domain heuristics or handcrafted features. The framework also includes a quantization tool called SeeDot [56].

CMSIS-NN: Cortex Microcontroller Software Interface Standard-NN [164] was designed to transform TF, PyTorch, and Caffe models for Cortex-M series MCU. It generates C++ files from the model, which can be included in the main program file and compiled. It consists of a collection of optimized neural network functions with fixed-point quantization, including fully connected layers, depth-wise separable convolution, partial image-to-column convolution, in-situ split x-y pooling, and activation functions (ReLU, sigmoid, and tanh, with the latter two implemented via lookup tables). It also features a collection of support functions including data type conversion and activation function tables (for sigmoid and tanh). CMSIS-NN provides 4.6× speedup and 4.9× energy savings over non-optimized convolutional models.

Edge Impulse EON Compiler: Edge Impulse [32] provides a complete end-to-end model deployment solution for TinyML devices, starting with data collection using IoT devices, extracting features, training the models, and then deployment and optimization of models for TinyML devices. It uses the interpreter-less **Edge Optimized Neural (EON) compiler** for model deployment, while also supporting TFLM. EON compiler directly compiles the network to C++ source code, eliminating the need to store ML operators that are not in use (at the cost of portability). EON compiler was shown to run the same network with 25-55% less SRAM and 35% less flash than TFLM.

STM32Cube.AI and NanoEdge AI Studio: X-Cube-AI from STMicroelectronics [172] generates STM32 compatible C code from a wide variety of deep-learning frameworks (e.g., PyTorch, TensorFlow, Keras, Caffe, MATLAB, Microsoft Cognitive Toolkit, Lasagne and ConvnetJS). It allows quantization (min-max), operator fusion, and the use of external flash or SRAM to store activation maps or weights. The tool also features functions to measure system performance and deployment accuracy and suggests a list of compatible STM32 platforms based on model complexity. X-Cube-AI was shown to provide 1.3× memory reduction and 2.2× speedup over TFLM for gesture recognition and keyword spotting [189]. NanoEdge AI Studio [173] is another AutoML framework from STMicroelectronics for prototyping anomaly detection, outlier detection, classification, and regression problems for STM32 platforms, including an embedded emulator.

Eloquent MicroML and TinyML: MicroMLgen ports decision trees, support vector machines (linear, polynomial, radial kernels or one-class), random forests, XGboost, Naive Bayes, relevant vector machines, and SEFR (a variant of SVM) from SciKit-Learn to Arduino-style C code, with the model entities stored on flash. It also supports onboard

feature projection through PCA. TinyMLgen ports TFLite models to optimized C code using TFLite's code generator [72].

Sklearn Porter: Sklearn Porter [174], generates C, Java, PHP, Ruby, GO, and Javascript code from Scikit-Learn models. It supports the conversion of support vector machines, decision trees, random forests, AdaBoost, k-nearest neighbors, Naive Bayes, and multi-layer perceptrons.

EmbML: Embedded ML [175] converts logistic regressors, decision trees, multi-layer perceptrons, and support vector machines (linear, polynomial, or radial kernels) models generated by Weka or Scikit-Learn to C++ code native to embedded hardware. It generates initialization variables, structures, and functions for classification, storing the classifier data on flash to avoid high memory usage, and supports the quantization of floating-point entities. EmbML was shown to reduce memory usage by 31% and latency by 92% over Sklearn Porter.

FANN-on-MCU: FANN-on-MCU [176] ports multi-layer perceptrons generated by fast artificial neural networks (FANN) library to Cortex-M series processors. It allows model quantization and produces an independent callable C function based on the specific instruction set of the MCU. It takes the memory of the target architecture into account and stores network parameters in either RAM or flash depending upon whichever does not overflow and closer to the processor (e.g., RAM is closer than flash).

SONIC, TAILS: Software-only neural intermittent computing (SONIC) and tile-accelerated intermittent low energy accelerator (LEA) support (TAILS) [24] are inference engines for intermittent computing systems. SONIC eliminates redo-logging, task transitions, and wasted work associated with moving data between SRAM and flash by introducing **loop continuation**, which allows loop index modification directly on the flash without expensive saving and restoring. To ensure idempotence, SONIC uses **loop ordered buffering** (loop reordering and double buffering partial feature maps to eliminate commits in a loop iterations) and **sparse undo-logging** (buffer reuse to ensure idempotence for sparse ML operators). SONIC introduces a latency overhead of only 25-75% over non-intermittent neural network execution (compared to 10× overhead from baseline intermittent model execution frameworks), reducing inference energy by 6.9× over competing baselines. TAILS exploits LEA in MSP430 microcontrollers to maximize throughput using direct-memory access and parallelism. LEA supports acceleration of finite-impulse-response discrete-time convolution. TAILS further reduces inference energy by 12.2× over competing baselines.

IX Online Learning

After deployment, the on-device model needs to be periodically updated to account for shifts in feature distribution in the wild [183]. While models trained on new data on a server could be sent out to the microcontroller once in a while, limited communication bandwidth and privacy concerns can prevent offloading the training to a server. However, the conventional training memory and energy footprint are much larger than the inference memory and energy footprint, rendering traditional GPu-based training techniques unsuitable

for microcontrollers [19]. Thus, several on-device training and federated learning (FL) frameworks have emerged for microcontrollers, summarized in Table IX and Table X.

On-device Training:

On-device training frameworks generally divide the learning process into three parts. *Firstly,* the training framework must be able to detect when a significant shift has happened in the input dataset (**when to learn**). This can be done by calculating the per-output covariate distribution divergence on principal feature components [183], running mean and variance of streaming input [184] or confidence score of predictions [186]. *Secondly,* the on-device training framework must perform model adaptation within device constraints and limited training samples (**how to learn**). Three key techniques have been proposed for on-device model adaptation.

(i) Last Layer Transfer Learning: The last layer of the network is fine-tuned through stochastic gradient descent one sample at a time [184] or reusing the outputs of feedforward execution without backpropagation [183] for batch gradient descent. Due to limited capacity and catastrophic forgetting, this approach results in poor performance when the distribution of new data is significantly different from the original training set [19].

(ii) Train Specialized Operators: TinyTL [19] proposed the use of lite residual learning modules for refining the output feature maps when updating just the bias instead of weights during on-device training to recoup performance loss. ML-MCU [185] proposed a lightweight one-versus-one (oVo) binary classifier for multiclass classification, which trains only those base classifiers that have a significant impact on final accuracy. This approach yields significant accuracy improvement over transfer learning (e.g. 34.1% higher than last layer transfer learning by TinyTL) without additional memory overhead but limits the application space due to constrained network types. Furthermore, TinyTL is not suitable for extremely resource-limited microcontrollers (e.g. Cortex-M).

(iii) Special Learning Techniques: Quantized continual learning prevents catastrophic forgetting by storing activation maps from past training data in the quantized form in a latent intermediate layer as replay data [188]. This allows learning from non-iiD data. incremental training uses constrained optimization to update the weights one sample at a time [186]. Both approaches suffer from limited application space due to limited supported network types. In addition, continual learning has high compute cost [188].

Thirdly, the training framework must be able to select the samples to pick for training to maximize the learning effect, especially to prevent catastrophic forgetting for transfer learning approaches (**what to learn**). Common techniques include selecting samples based on their gradient norm, oversampling minority classes, and using weighted replay or sample importance weighing [183] [187]. Unfortunately, none of the on-device training frameworks are directly compatible with popular TinyML software suites, as none of the software suites are capable of unfreezing the frozen model graph on board. Moreover, all on-device training frameworks only work with networks having simple and limited architectural choices to

prevent resource overflow. Thereby, additional memory constraints need to be injected into NAS frameworks to limit the model complexity.

Federated Learning:

FL extends on-device training to a distributed and non-iiD setting, where the edge devices update parameters of a shared model on board, send the local versions of the updated model to a server, and receive a common and robust aggregated model, without the data ever leaving the edge devices [190]. We compare different FL frameworks suitable for TinyML listed in Table X using five distinguishing properties:

(i) FL Strategy: FL strategy refers to the selection of FL algorithms the frameworks provide. Most FL frameworks provide vanilla federated averaging (FedAvg) algorithm, where the local model weights are aggregated at the server instead of the gradients for communication efficiency [198]. Several enhancements to FedAvg have emerged to handle heterogeneity and resource constraints of AI-IoT devices. These include variants that have the following properties:

- Robust to laggards or client disconnections [190].
- Achieves similar accuracy across all devices [190].
- Includes device-specific model pruning to improve communication and training cost [192] [194].
- uses transfer learning or fine-tuning for local model updates to save memory and build personalized models [195] [197].
- uses knowledge distillation to aggregate class probabilities instead of weights [197].

Wu *et al.* [197] showed that transfer learning and knowledge distillation variants provide 5-11% accuracy improvement over vanilla FedAvg for human action recognition, while providing 10-5000× reduction in communication cost. Pang *et al.* [196] proposed the use of RL for model aggregation, obtaining 1.4-2.7% higher accuracy over FedAvg for image recognition.

(ii) Communication Stack: FL requires a robust and efficient communication stack between the server and edge devices. Most FL frameworks rely on the robustness and efficiency guarantees provided by FedAvg and other FL strategies, such as the use of pruning or knowledge distillation over class probabilities [192] [194] [197]. Flower [190] and DIoT [193] provide bidirectional gRPC and WebSocket protocols to provide low-latency, concurrent, and asynchronous communication between server and clients. Both protocols are language, serialization, and communication agnostic.

(iii) Scalability and Heterogeneity: FL frameworks must be able to run workloads on hardware with different compute and communication budgets in a scalable fashion. *First*, the frameworks must be able to detect and track resource and task completion measures. Flower [190] includes a virtual client engine for scheduling and resource management. FedPARL [192] provides a resource and trust value tracker to monitor resource availability, bandwidth,

task completion, task delay, and model integrity. DIoT [193] uses an unsupervised learning method to identify device state and type based on network traffic. *Secondly*, the frameworks should have a course of action for optimal workload distribution among these clients based on detected measures. Proposed techniques include partial work (average model weights based on gradient update sample count instead of timeout threshold) [190] [192], importance sampling (improve client selection probability of least-contributing clients) [190], adaptive pruning [194], and RL-based automated collaboration scheme discovery [196]. *Thirdly*, the proposed techniques must generalize to a large number of clients. Among the different FL TinyML frameworks, Flower [190] has been shown to scale to 15M clients (1000 concurrent clients).

(iv) Privacy: FL frameworks must ensure that the local or global models cannot be reverse-engineered to uncover client data. Most federated learning frameworks for TinyML rely on the assumption that weight updates cannot be reverse-engineered to uncover local data. However, membership inference [203] and model inversion attacks [204] are successful against vanilla FedAvg. As a result, Flower [190] proposed using secure aggregation in their framework instead of vanilla model aggregation. The proposed semi-honest protocol is robust against client dropouts, uses a multiparty computation protocol that does not require trusted hardware, and has low compute and communication overhead [205].

(v) Client Hardware and Supported Languages: Finally, the FL frameworks must support a wide variety of clients with different processors and operating languages. Among the different frameworks, Flower [190], PruneFL [194], and TinyFedTL [195] were tested on microcontrollers, supporting Python, Java, and C++.

X Key Applications

Depending on the application, several variants of the TinyML workflow are used. In this section, we provide application-specific numerical insights from these workflows.

Image Recognition and Visual Wake Words:

Since the inception of AlexNet in 2012 [93], deep neural networks have been extensively used for visual understanding, such as image classification, object detection, handwriting recognition, visual wake words detection, and semantic segmentation [14] [206]. The trend has trickled into the TinyML community as well, evident in Table XI and Table XII. Image recognition on the CIFAR-10 dataset and person detection on the Visual Wake Words dataset are two inference benchmarks in the MLPerf Tiny v0.5 [9]. Among the techniques shown in Table XI, NAS on residual convolutional architectures (e.g., MCUNetV2 [121], μNAS [122], and SpArSe [86]), rapid downsampling (e.g., RNNpool [60]), sparse projection (e.g., Bonsai [57] and ProtoNN [58]) and deep compression (e.g., Compressed LeNet [49] and SqueezeNet [92]) are the most common. Models that operate on multiclass datasets are suitable for cortex M class architectures, while models that operate on binary datasets have been shown to be deployable on AVR RISC microcontrollers. In Table XI, MCUNetv2 [121] and AttendNets [114] achieved the state-of-the-art top 1% accuracy (72-73%) on ImageNet for microcontrollers. MCUNetv2 uses once-for-all NAS on convolutional

operators, combined with patch-by-patch inference and receptive field redistribution during runtime [121]. AttendNets uses a standalone visual attention condenser, which improve spatial-channel selective attention [114]. MCUNetv2 [121] (with a YOLOv3 backbone) and AttendSeg [115] (with attention condensers) achieved state-of-the-art performance for semantic segmentation on the Pascal VOC dataset and CamVid datasets, respectively. μNAS CNN achieved the state-of-the-art performance on CIFAR-10 and MNIST [122]. Two interesting applications that deviate from traditional machine vision datasets include American sign language prediction [200] and detecting face masks in light of COVID-19 [201].

Detecting visual wake words (i.e. person detection) is a special case of image recognition. Table XII lists some of the models proposed for performing wake words detection on the visual wake words dataset [14]. Among all the proposed models, RaScaNet [202] achieves the best balance of accuracy and resource usage. RaScaNet extracts features from an image patch using convolutional blocks, and then sequentially learns the latent representation of the entire image using recurrent blocks. The network also includes both spatial and channel attention to focus spatially distinct and multi-head discriminative feature representations.

Audio Keyword Spotting and Speech Recognition:

Voice is a core component in human-computer interaction. Audio keyword spotting or wake-word detection are used in voice assistants to identify waking keywords (e.g., ”Hey Google™”, ”Hey Siri™” or ”Alexa™”). The assistants must continuously listen for the keyword in utterances without being power or resource hungry [9]. Table XIII lists some keyword spotting, speech enhancement, and wake-words detection models geared towards microcontrollers. The use of lightweight depthwise-separable convolution, attention condensers, and recurrent units have generated models that are in the order of 10^0 kB. Some of the models (e.g., FastGRNN, ShallowRNN, and ULP RNN) can even run on AVR RISC microcontrollers with 2 kB SRAM, while others are suitable for deployment on Cortex M class microcontrollers. The models typically operate on log Mel-spectrograms, which are short-time Fourier transforms transferred to the Mel scale [210] and available on CMSIS-DSP library for embedded implementation. Most models use the Google Speech Commands Dataset for training, which has 35 words with 105,829 utterances from 2,618 speakers [10].

Anomaly Detection:

Anomaly detection or one-class classification detects outliers or deviations in the input data stream to indicate malfunctions [120] in an unsupervised fashion. Included in MLPerf Tiny v0.5 benchmark, applications of anomaly detection include diagnosis of industrial machinery [9] [120] [8], physiological disorders (e.g., heart attacks, seizures, etc.) [171], and climate conditions [211]. The two most common network architectures for microcontroller-based anomaly detection are fully-connected autoencoders (FC-AE) and depthwise CNN. Table XIV lists some of the anomaly detectors developed for microcontrollers. Among the different techniques, DROCC can operate directly on raw audio, sensor data, and images [171] without feature extraction. DROOC assumes that normal points lie on a low-dimensional linear manifold while points surrounding the normal points outside a threshold

radius are outliers, which can be augmented in a generative adversarial manner into the training set. other audio-based anomaly detectors generally operate on mel-spectrograms [9] [120] [8].

Activity and Gesture Tracking:

Activity and gesture tracking form the central oracle for many applications, including health monitoring, behavioral analysis, context detection, augmented reality, and speech recognition [2]. Table XV showcases some activity detection framework geared towards microcontrollers. The common theme is to use lightweight models or use conventional models with a lower number of layers or polynomial complexity. Most models achieve accuracies of 90% or more for simple macro-gestures (e.g., discrete fist gestures) or macro-activities (e.g., walking, running, standing, turning, jumping), while being 10^0 or 10^1 kB order of size. The models are mostly hand-tuned due to the innate lightweight nature of the chosen models, with a few automated using NAs.

Odometry and Navigation:

Odometry is the fusion of onboard sensors for indirect estimation of an object's position and attitude under absence or in conjunction with infrastructure-dependent localization services [218]. TinyOdom [1] exploits THIN-Bayes, TCN backbone, and a magnetometer, physics, and velocity-centric sequence learning formulation to yield neural inertial odometry models that are 31-134× smaller than existing neural inertial odometry models, suitable for deployment on Cortex-M architectures. Vehicle neural networks (VNN) [21] use a modified and quantized LeNet-5 as an autonomous controller on a car under stochastic lighting conditions. The network leverages imitation learning via a classical computer vision teacher algorithm for training. VNNs have 7.5-163.5 kMACs and the PULP implementation on GAP8 SoC reduces latency and energy consumption by 13× (0.2-1.2 mS) and 3.2× (3.9-18.9 μJ per inference), respectively over Cortex-M architectures, achieving 97% accuracy. A class of residual networks intended for standard-uAV navigation without SLAM called DroNets [219] have been ported on nano-UAVs retrofitted with a PULP GAP8 SoC shield [220]. By using tiling, quantization, parallelization, and signal-processing on the PULP chip, the platform achieved 6-18 FPS within a 64 mW power envelope, covering 113 m unseen indoor trajectory in the real world at a speed of 1.5 m/s [22]. In all cases, the odometry models enjoy the lightweightness of classical odometry techniques and the resolution of large networks.

mHealth:

TinyML opens up a broad spectrum of real-time and low-footprint eHealth applications, some of which are summarized in Table XVI. These include monitoring eating episodes and coughs using microphones [221] [223], sleep monitoring and arrythmia detection through ECG measurements [222] [171], epileptic seizure recognition from EEG sensors [171], and fall detection using earable inertial sensors [2]. Most TinyML mHealth applications are variants of anomaly detection, indicating presence or absence of a health condition, thereby allowing use of ultra lightweight models in the order of 10^0 to 10^1 kB. Example models for mHealth include Bonsai [2], embedded GRU [221], 1D CNN [222], FC-AE [171], and 2-layer CNN/LSTM [223].

Facial Biometrics:

Facial biometrics has been a prominent authentication technique in civilian and military applications [226]. Existing face detectors use deep-learning to automate the feature representation pipeline while approaching human performance [226]. Table XVII illustrates some face detectors for microcontrollers. A common recipe for porting deep face detectors onto microcontrollers includes the use of lightweight spatial convolution coupled with NAS, quantization, and inference engine optimizations. Typical neural blocks include squeeze and excitation modules [224], coupling depthwise with pointwise convolution [225], and non-linear pooling between convolutional layers [60]. Successive and rapid downsampling helps cut out redundant network layers further [60] [225] while ensuring scale-equitable face detection. Inference engine optimizations include patch-based inference scheduling [121], receptive field redistribution [121], and dual memory management [224]. Patch-by-patch inference allows operation on only a tiny region of the activation map, while receptive field redistribution shifts receptive field and FLoPS to later stages to mitigate peak memory usage and overlapping patches [121]. Dual memory management swaps variables between flash and RAM whenever required [224].

XI Discussion and Case Studies

In this section, we break down the end-to-end workflow and provide quantitative analysis of individual aspects of the workflow based on select examples from section IV to Section X. We discuss how individual aspects contribute to the overall execution, and also describe qualitatively how individual techniques for one aspect impact the choices for other aspects.

Feature Projection vs. No Feature Projection:

Feature projection allows a domain expert to retain data variance while reducing data dimensionality [2]. Intuitively, this reduces the model complexity needed to capture the variations in input data, i.e., feature projection is useful for simplifying the architecture of non-TinyML models. Consider the gesture recognition example in Table XVIII. Both CNN and the MLP achieve the same accuracy. However, the MLP pipeline, operating on spectral features, requires 2.5× less flash, runs 2.2× faster, and requires 18× less SRAM than the CNN pipeline, which operates on raw data. By leveraging domain knowledge, simpler models can achieve the same accuracy, yet save memory and inference costs over complex models. Well-designed features (e.g., audio MFCC, spectrograms, and signal power) are also able to exploit DSP functions (e.g. CMSIS-DSP) and accelerators embedded in most microcontrollers [6]. However, if the extracted features are not sufficiently discriminatory due to a lack of domain knowledge, then the performance of models will degrade [227]. Consider the human activity recognition example in Table XVIII. Bonsai operates on five statistical features surrounding the signal amplitude, which are unable to sufficiently distinguish among activity primitives that are statistically similar (e.g., sitting and sleeping). Thus, Bonsai suffers a 17% accuracy drop while being similar in size to FastGRNN. To achieve the performance of complex models that do not operate on features while having the computational efficiency of simple models operating on handcrafted features, ultra-lightweight ML blocks are used. These blocks can often be much more efficient and accurate than models tied to a feature extraction pipeline, as poorly designed features can

yield significant compute overhead [2]. For example, in Table XVIII, DROCC outperforms one-class sVM for anomaly detection not only in accuracy (+20% gain) but also in model size (1600× reduction). Unfortunately, the problem with lightweight models is two-fold. *Firstly,* most of these models do not have enough parameters to model globally significant features, failing to generalize to new data distributions in the field [2]. *Secondly,* most NAS frameworks, TinyML software suites, intermittent computing tools, and online learning frameworks lack support for deploying some of these models on commodity microcontrollers. Therefore, adopting feature extraction requires careful understanding of the application constraints, striking a balance between the availability of domain knowledge, feasible model architecture sets, feature acceleration support, and support from model optimization and architecture search tools.

Compression vs. No Compression:

Exploiting sparsity and reducing bitwidth of models depends on three key factors:

(i) Large-Sparse vs. Small-Dense: Large-sparse models (compressed and vanilla models) are known to outperform small-dense models (uncompressed and lightweight models) for a broad range of network architectures [48] in terms of compression ratio for comparable accuracy. This is evident from the speech commands and MNIST-10 examples in Table XIX. LSTM-Prune and LSTM-KP outperform FastGRNN and Bonsai, providing on average 12× model size reduction with only 2.3% accuracy loss. Moreover, on average, all uncompressed and vanilla models provided a 13.5× reduction in model size when pruned, compared to 2.1× for lightweight models (FastGRNN). Therefore, sparsification is useful when working with vanilla models rather than lightweight ML blocks.

(ii) Pruning and Quantization Gains: Unstructured pruning and post-training quantization offer performance gains in different dimensions. Generally, both pruning and quantization are applied jointly [49].

Flash savings: Pruning is more aggressive in reducing the model size than quantization [49]. In Table XIX, on average, pruning provides 13.6× compression factor, compared to 3.9× compress factor provided by quantization. Pruning combined with quantization provides a 16× reduction in model size on average.

SRAM savings: Pruning is less likely to reduce working memory footprint than quantization. After pruning, the microcontroller still has to perform multiplication in the original floating-point bitwidth, whereas in quantization, the bitwidth of the multiplication decreases.

Latency: Intuitively, pruning is less likely to reduce the inference latency compared to integer quantization in microcontrollers. The gains from the loss of redundant weights are lower than the gains from the integer matrix multiplication. Moreover, unstructured pruning can add processing and execution time overhead [85].

Accuracy loss: Pruning often causes higher accuracy loss than quantization. In Table XIX, on average, pruning reduced accuracy by 4.9%, compared 0.4% from quantization. This is

due to a higher degree of information loss in pruning as in quantization only the bit-width is reduced.

(iii) Support from HW/SW: Not all microcontrollers and TinyML software suites support or can reap the benefits of quantization of intermediate or sub-byte bitwidth [81]. For example, TFLM does not support arbitrary bitwidth of weights and activations [55]. Most microcontrollers are limited by their SIMD bitwidth, unable to exploit low precision representation of neural networks fully [81]. Therefore, care must be taken to ensure the chosen quantization scheme is compatible with the choice of microcontroller and TinyML software suites.

Lightweight Models vs. Vanilla Models:

Most model compression techniques cannot reduce the size of pre-trained models without significant loss in accuracy (e.g. pruning and quantization results in 19× reduction in model size on average in Table XIX). In some cases, the pre-trained model is too big to apply model compression feasibly for a microcontroller (e.g., in Table XIX, AlexNet can be reduced to 6.9 MB from 240 MB) or the pre-trained model may not even be a neural network (e.g., in Table XX, SVM, Coarse DT, AdaBoost, and kNN are non-neural models). In such cases, lightweight ML blocks are adopted to reduce the model size and inference latency while maintaining or exceeding the accuracy of vanilla models. In fact, from Table XX, we see that lightweight ML blocks are commonly adopted when out-of-memory errors are encountered on the microcontroller. For the human activity recognition (Auritus) and image recognition (MNIST-10 and ImageNet) use cases, the vanilla models (SVM, MLP, Coarse DT, AdaBoost, AlexNet) were simply too big to run on commodity microcontrollers, forcing the adoption of lightweight ML operators (Bonsai, ProtoNN, TCN, AttendNets, SqueezeNet). In some cases, lightweight models are adopted to improve the accuracy and latency (e.g., FastGRNN has higher accuracy and lower latency than RNN). However, special attention must be paid to the specific compute budget when adopting these lightweight models. *Firstly,* some of these models might improve the metrics in one dimension and degrade other dimensions. For example, in Table XIX and Table XX, SqueezeNet has lower model size but higher latency (and energy usage) than AlexNet [230]. *Secondly*, as discussed earlier in the feature projection case study, some lightweight models overfit the training set and fail to generalize to unseen data. For example, in Table XX, TCN has a 5% reduction in test accuracy over SVM, MLP, Coarse DT, and AdaBoost. In fact, for activity detection, Saha *et al.* [2] showed that lightweight ML blocks have an accuracy drop of 11.8% for the same test set distribution shift over vanilla models. *Thirdly*, not all aspects of the TinyML workflow support every lightweight ML block. For example, μNAS [122], MicroNets [8], and SpArSe [86] assume a CNN backbone, while Sklearn Porter [174] only supports porting MLP to microcontrollers. Moreover, most on-device learning frameworks only support CNN backbones. Thus, the choice of lightweight ML blocks is limited by what the other components in the TinyML workflow support.

Using NAS vs. Handcrafted Models:

NAS is used when one or more model performance metrics (e.g., latency, SRAM usage, energy) need to be constrained to suit the deployment scenario. NAS is particularly useful in three cases:

(i) Metrics Form Competing Objectives: The most common motivation behind NAS is to increase the model accuracy while decreasing the flash, SRAM, latency, and energy usage. These metrics form competing objectives under search space and device constraints. For example, a larger model is likely to provide higher accuracy but consume more flash and SRAM. A certain architecture (e.g., SqueezeNet) is likely to reduce flash usage but can have higher latency than a larger model (e.g. AlexNet). The model might have to follow certain bounds or rules (e.g., cannot use a specific operator type). In Table XXI, the goal is to find the best performing models that reach the desired objectives within the specified constraints. In all cases, the NAS strategy consistently outperforms handcrafted models in terms of providing the most accurate model within the device constraints.

(ii) Optimize High-Dimensional Search Space for Multiple Target Hardware: Neural network search spaces can grow intractable quickly. For example, the search space of a CNN can contain the number of layers, the number of kernels in each layer, the size of the kernel in each layer, the stride in each layer, the size of kernels in the pooling layer, etc [122]. The search space might even contain parameters for different model architectures. Furthermore, the network might have to be optimized for multiple microcontrollers with distinct compute and memory budget [5]. To save human time and effort, NAS algorithms can automatically perform model architectural adaption to fully exploit the target capabilities of different hardware. In the inertial odometry example in Table XXI, TinyOdom [1] produces four different models that provide a competitive resolution within the memory constraints of four different microcontrollers, providing a 1.6-30× reduction in model size while suffering a resolution drop of 1.2× compared to handcrafted models. Similarly, in the keyword spotting example in Table XXI, MicroNets [8] generates three different DS-CNN that are suitable for three different microcontroller models, outperforming the handcrafted DS-CNN by 3.3-4.5%.

(iii) Prior Wisdom Does Not Suit Deployment Needs: In some deployment scenarios, expert knowledge may not suit the deployment needs. For example, in the inertial odometry case in Table XXI, Tinyodom [1] was the first framework allowing the deployment of inertial odometry models on microcontrollers. In the case of image recognition, μNAS [122] attempted to deploy the models on AVR RISC microcontrollers, which have a much tighter resource budget than Cortex M4 microcontrollers used by Bonsai [57] and SpArSe [86]. Similarly, SpArSe [86] attempted to run deep neural networks on microcontrollers and not non-neural models to broaden the application spectrum of AI-IoT. Under unexplored circumstances, NAS can bring in valuable insights during model discovery on achievable performance and optimal architectural choices.

Using Runtime Optimizations vs. No Optimizations:

The use of TinyML software suites to generate code and perform operator/inference engine optimizations is a mandatory step in the TinyML workflow, often needed to guarantee the execution of a trained model on the microcontroller. Consider the CIFAR-10 image recognition example in Table XXII. The use of partial *im2col* in CMSIS allows the CNN to have a working memory of 133 kB instead of 332 kB, in which case the CNN would overflow the Cortex-M7 SRAM [164]. The optimized operator set also reduces the inference latency by 4.6× and decreases energy usage by 4.9× [164]. Similarly, MCUNetv2 [121] achieved record ImageNet and Pascal VOC accuracy on microcontrollers by optimizing a large MBNetv2 that normally overflows the SRAM using patch-by-patch inference and receptive field redistribution. However, to pick the appropriate software suite, other questions must be asked.

1. Which microcontrollers are suitable for my application?
2. What are the memory, latency, and energy requirements?
3. Which ML blocks are suitable for my application?
4. Which training frameworks can I use?
5. Do I need support for intermittent computing?
6. Do I need support for online learning?
7. Do I need an automated schedule explorer?
8. Is dynamic memory management necessary?
9. How many models need to run on the same platform?
10. Do I need to share the same model across platforms?
11. Do I need to sparsify or quantize any model?

Consider the human activity recognition and keyword spotting use case in Table XXII. TFLM uses an interpreter-based approach to realize the model graph during runtime [167]. TFLM supports dynamic memory management {7)}, multitenancy {8)}, and updating the model binaries rather than the entire codebase for fast prototyping and portability across platforms {9)}. However, {7)}, {8)} and {9)} come at 1.3× increase in flash usage and 2.6× increase in latency (lagging on {2)}) compared to STM32Cube.AI, which embeds operator function calls into native C code [172]. STM32Cube.AI, on the other hand, only supports STM32 series of Cortex-M microcontrollers (lagging on {1)}), while TFLM provides much broader platform support. However, STM32Cube.AI also supports non-neural model deployment (e.g., k-means, SVM, RF, kNN, DT, etc.), while TFLM only supports neural network deployment (lagging on {3)}). Likewise, {5)} can only be realized through SONIC, TAILS [24], and {7)} is provided by only micro-TVM [130]. Both quantitative and qualitative tradeoffs similar to the case study here must be performed to pick the appropriate software suite.

Using Online Learning vs. Static Models:

Online learning improves the performance of the model by adapting the model on board without sensitive data leaving the device. Consider the case studies on online learning shown in Table XXIII. The performance of models improves by 34% when on-device training is used to adapt to dataset shifts. For TinyOL, the latency overhead to include online learning is 10%. While the performance gains are somewhat transparent, the major barrier in on-device learning is the lack of support from other aspects of the workflow. For example, most on-device learning frameworks assume the use of CNN or binary classifiers and also use a custom code generator for the model due to a lack of online learning support from existing TinyML software suites. Moreover, it is not clear how NAS should account for the training memory and inference overheads when on-device learning is used. The lack of comprehensive studies of on-device learning also limits the adoption of federated learning in TinyML. Particularly, while the TinyML workflow was designed for a single non-collaborative model, federated learning requires the distribution of a global model to be enhanced via local model updates. While existing federated learning frameworks have tools to distribute resources heterogeneously, it is unclear how NAS, model compression, or lightweight ML blocks affect the real-world setting, as none of the federated learning frameworks have studied these effects. Thereby, online learning constrains the user to a very specific choice of models and custom software suites.

XII Challenges and Opportunities

The first-generation efforts in TinyML focused on the engineering and mechanics of squeezing ML models within the limited memory, compute, and power bounds of a microcontroller. Both academia and industry have established several TinyML software frameworks to streamline the deployment of ML models for microcontrollers. Many of the issues raised by prior surveys [3] [4] [6] have been addressed. However, the following new challenges are emerging that require further research.

Application Specific Safety and Heuristic Requirements:

Real-world IoT applications operate within certain bounds, correlations, and heuristic rules set forth by the operating domain and system physics. For example, a UAV cannot exceed a certain bank angle without compromising stability [233]. In complex event processing, specific granular action primitives (e.g., cooking a dish) must always precede other primitives (e.g., chopping vegetables) [234]. Neural networks cannot assure that the learned distributions obey all the laws [235]. As a result. recent neural network pipelines are being injected with trainable neuro-symbolic reasoning [236] [237], signal temporal logic [235], and physics-aware embeddings [238] [239] [240] [241] for robust complex event processing within the laws and bounds of physics. For making rich and complex inferences beyond binary classification, the TinyML workflow requires research to combine data and human knowledge by including logical reasoning modules within the microcontroller’s compute and memory bounds.

Data Quality and Uncertainty Awareness:

Sensor data in the wild suffers from missing data, cross-channel timestamp misalignment, and window jitter [227] [242]. These uncertainties may stem from scheduling and timing stack delays, system clock imperfections, sensor malfunction, memory overflow, or power constraints [243] [244]. Sensing uncertainty can reduce the performance of ML models when training for complex event processing [227]. TinyML models need to be injected with uncertainty awareness by incorporating appropriate training frameworks [227] [242] in the workflow or use onboard clocks and hardware enhancements for precise time-synchronization [245].

On-Device Fine-Tuning:

Models in the wild need to be fine-tuned periodically to ensure robustness across domain shifts in incoming data distribution [183]. *Firstly*, while several on-device learning frameworks have been proposed for edge devices [246], they either work on high-end edge devices (e.g., Raspberry Pi) [19] [247] or can update weights of a few layers on microcontrollers [183]. Software-centric resource constraints, constrained learning theories, and static resource budget prevent on-device learning from being a viable alternative to cloud-based training for microcontrollers [246]. *Secondly,* an alternate line of work suggests low-latency compressive offloading onto the cloud [20] but has non-deterministic compression ratios and offloading points. *Lastly,* the models themselves can be made more robust to domain shifts through representation learning [248] or domain-adversarial training [249], but the resulting models do not fit on microcontrollers. More work needs to be done in striking an optimal balance between on-device fine-tuning and over-the-air model updates, and whether unsupervised embeddings can be ported onto microcontrollers.

Backward Compatibility:

The changes in behavior when deploying an upstream model (e.g., a model on the cloud) to microcontrollers through the TinyML workflow cannot be measured in isolation using only the aggregate performance measures (such as accuracy) [250] Even when a TinyML model (downstream model) and the upstream model have the same accuracy, they may not be functionally equivalent and may have sample-wise inconsistencies [251] resulting in new failures impacting high stake domains such as healthcare. This notion of functional equivalence between an upstream and a downstream model is known as backward compatibility. When previously unseen errors are observed in the downstream model, the downstream model is said to be backward incompatible [252] and has low fidelity [253] and high perceived regression [251] with respect to the upstream model. As a result, to have robust inference, the TinyML model must have both high accuracy and high fidelity with its upstream counterpart. Proposed solutions, such as positive congruent training [251] and backward compatible learning [254], are yet to be integrated and optimized for the TinyML workflow.

New Security and Privacy Threats:

While constraining private data within the IoT node reduce the chance of privacy and security leaks associated with cloud-based inference, the attack surface on TinyML

platforms is wide open. Compressed models are prone to adversarial attacks and false data injection with a higher success rate than larger models [255] [256] [257]. At the sensing layer, microarchitectural and physical side channels can leak information from microcontroller chips through cache leaks, power analysis, and electromagnetic analysis [258]. Direct attacks on IoT devices include malware injection, model extraction, access control, man-in-the-middle, flooding, and routing [258]. Therefore, the NAS optimization function in the TinyML workflow should include adversarial robustness goals to provide not only the smallest models but also the models most robust to adversarial attacks [256] [257] [259]. The workflow should also include attack surface analysis and tools to defend the inference pipeline against attacks.

Hardware/Software Co-Exploration:

Much of the development in TinyML has been software-driven, with the hardware platform being static. While IoT platforms hosting microcontrollers are shrinking due to Moore's Law, the workload and the complexity of neural networks have skyrocketed [7] [260]. Proposed hardware innovations include the use of a systolic array, stochastic computing, in-memory computing, near-data processing, spiking neural hardware, and non-von Neumann architectures [7] [260] [261]. However, such architecture innovations are largely disjoint from the TinyML software communities. Developments in TinyML software need to be performed hand-in-hand with attention-directed hardware design, with the platform and model being optimized jointly [262] [263].

XIII. Conclusion

It is desirable to enable onboard ML on microcontrollers, turning them from simple data harvesters to learning-enabled inference generators. To that end, we introduced a widely applicable workflow of ML model development and deployment on microcontroller class devices. Several applications are showcased to highlight the tradeoffs in different instances of this workflow adoption. Although the current efforts can transition the state-of-the-art ML models to ultra resource-constrained environments, we consider them as the first generation of TinyML and present new opportunities. Through this review, we envision a need for the next generation of TinyML frameworks to address the discussed challenges that have received limited explorations.

Acknowledgments

This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; by the IoBT REIGN Collaborative Research Alliance funded by the Army Research Laboratory (ARL) under Cooperative Agreement W911NF-17-2-0196; and by the NIH mHealth Center for Discovery, Optimization and Translation of Temporally-Precise Interventions (mDOT) under award 1P41EB028242.

References

[1]. Saha SS, Sandha SS, Garcia L, and Srivastava M, "Tinyodom: Hardware-aware efficient neural inertial navigation," in Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies. ACM New York, NY, USA, 2022.

[2]. Saha SS, Sandha SS, Pei S, Jain V, Wang Z, Li Y, Sarker A, and Srivastava M, “Auritus: An open-source optimization toolkit for training and development of human movement models and filters using earables,” in Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies. ACM New York, NY, USA, 2022.

[3]. Sanchez-Iborra R and Skarmeta AF, “Tinyml-enabled frugal smart objects: Challenges and opportunities,” IEEE Circuits and Systems Magazine, vol. 20, no. 3, pp. 4—18, 2020.

[4]. Dutta L and Bharali S, “Tinyml meets iot: A comprehensive survey,” Internet of Things, vol. 16, p. 100461, 2021.

[5]. Cai H, Gan C, Wang T, Zhang Z, and Han S, “Once-for-all: Train one network and specialize it for efficient deployment,” in International Conference on Learning Representations, 2019.

[6]. Ray PP, “A review on tinyml: State-of-the-art and prospects,” Journal of King Saud University-Computer and Information Sciences, 2021.

[7]. Shafique M, Theocharides T, Reddy VJ, and Murmann B, “Tinyml: Current progress, research challenges, and future roadmap,” in 2021 58th ACM/IEEE Design Automation Conference (DAC). IEEE, 2021, pp. 1303–1306.

[8]. Banbury C, Zhou C, Fedorov I, Matas R, Thakker U, Gope D, Janapa Reddi V, Mattina M, and Whatmough P, “Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers,” Proceedings of Machine Learning and Systems, vol. 3, 2021.

[9]. Banbury C, Reddi VJ, Torelli P, Holleman J, Jeffries N, Kiraly C, Montino P, Kanter D, Ahmed S, Pau D et al. , “Mlperf tiny benchmark,” Advances in Neural Information Processing Systems, 2021.

[10]. Warden P, “Speech commands: A dataset for limited-vocabulary speech recognition,” arXiv preprint arXiv:1804.03209, 2018.

[11]. Chollet F, “Xception: Deep learning with depthwise separable convolutions,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 1251–1258.

[12]. Howard AG, Zhu M, Chen B, Kalenichenko D, Wang W, Weyand T, Andreetto M, and Adam H, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” arXiv preprint arXiv:1704.04861, 2017.

[13]. Zhang Y, Suda N, Lai L, and Chandra V, “Hello edge: Keyword spotting on microcontrollers,” arXiv preprint arXiv:1711.07128, 2017.

[14]. Chowdhery A, Warden P, Shlens J, Howard A, and Rhodes R, “Visual wake words dataset,” arXiv preprint arXiv:1906.05721, 2019.

[15]. Krizhevsky A, “Learning multiple layers of features from tiny images,” Master’s thesis, University of Tront, 2009.

[16]. He K, Zhang X, Ren S, and Sun J, “Deep residual learning for image recognition,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.

[17]. Koizumi Y, Saito S, Uematsu H, Harada N, and Imoto K, “Toyadmos: A dataset of miniature-machine operating sounds for anomalous sound detection,” in 2019 IEEE WKSH on Applications of Signal Proceedings to Audio and Acoustics (WASPAA). IEEE, 2019, pp. 313–317.

[18]. Purohit H, Tanabe R, Ichige K, Endo T, Nikaido Y, Suefusa K, and Kawaguchi Y, “Mimii dataset: Sound dataset for malfunctioning industrial machine investigation and inspection,” in Acoustic Scenes and Events 2019 WKSH (DCASE2019), 2019, p. 209.

[19]. Cai H, Gan C, Zhu L, and Han S, “Tinytl: Reduce memory, not parameters for efficient on-device learning,” Advances in Neural Information Processing Systems, vol. 33, 2020.

[20]. Yao S, Li J, Liu D, Wang T, Liu S, Shao H, and Abdelzaher T, “Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency,” in Proceedings of the 18th Conference on Embedded Networked Sensor Systems, 2020, pp. 476–488.

[21]. de Prado M, Rusci M, Capotondi A, Donze R, Benini L, and Pazos N, “Robustifying the deployment of tinyml models for autonomous mini-vehicles,” Sensors, vol. 21, no. 4, p. 1339, 2021. [PubMed: 33668645]

[22]. Palossi D, Conti F, and Benini L, “An open source and open hardware deep learning-powered visual navigation engine for autonomous nanouavs,” in 2019 15th International Conference on Distributed Computer in Sensor Systems (DCOSS). IEEE, 2019, pp. 604–611.

[23]. Xie Z, Berseth G, Clary P, Hurst J, and van de Panne M, “Feedback control for cassie with deep reinforcement learning,” in 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2018, pp. 1241–1246.

[24]. Gobieski G, Lucia B, and Beckmann N, “Intelligence beyond the edge: Inference on intermittent embedded systems,” in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 199–213.

[25]. Giordano M, Mayer P, and Magno M, “A battery-free long-range wireless smart camera for face detection,” in Proceedings of the 8th International WKSH on Energy Harvesting and Energy-Neutral Sensing Systems, 2020, pp. 29–35.

[26]. Lee S, Islam B, Luo Y, and Nirjon S, “Intermittent learning: On-device machine learning on intermittently powered system,” Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, vol. 3, no. 4, pp. 1–30, 2019. [PubMed: 34164595]

[27]. Li T, Sahu AK, Talwalkar A, and Smith V, “Federated learning: Challenges, methods, and future directions,” IEEE Signal Proceedings Magazine, vol. 37, no. 3, pp. 50–60, 2020.

[28]. Bonawitz K, Ivanov V, Kreuter B, Marcedone A, McMahan HB, Patel S, Ramage D, Segal A, and Seth K, “Practical secure aggregation for privacy-preserving machine learning,” in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communication Security, 2017, pp. 1175–1191.

[29]. Naehrig M, Lauter K, and Vaikuntanathan V, “Can homomorphic encryption be practical?” in Proceedings of the 3rd ACM WKSH on Cloud Computer security WKSH, 2011, pp. 113–124.

[30]. Schmidhuber J, “Deep learning in neural networks: An overview,” Neural Networks, vol. 61, pp. 85–117, 2015. [PubMed: 25462637]

[31]. Lin J, Chen W-M, Lin Y, Gan C, Han S et al. , “Mcunet: Tiny deep learning on iot devices,” Advances in Neural Information Processing Systems, vol. 33, pp. 11 711–11 722, 2020.

[32]. “Edge impulse.” [Online]. Available: https://www.edgeimpulse.com/

[33]. Mazumder M, Chitlangia S, Banbury C, Kang Y, Ciro JM, Achorn K, Galvez D, Sabini M, Mattson P, Kanter D et al., “Multilingual spoken words corpus,” in Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2), 2021.

[34]. “Sensiml: Making sensor data sensible.” [Online]. Available: https://sensiml.com/

[35]. “Qeexo automl.” [Online]. Available: https://qeexo.com/

[36]. “Plumerai.” [Online]. Available: https://plumerai.com/

[37]. Roh Y, Heo G, and Whang SE, “A survey on data collection for machine learning: a big data-ai integration perspective,” IEEE Transactions on Knowledge and Data Engineering, vol. 33, no. 4, pp. 1328–1347, 2019.

[38]. Landset S, Khoshgoftaar TM, Richter AN, and Hasanin T, “A survey of open source tools for machine learning with big data in the hadoop ecosystem,” Journal of Big Data, vol. 2, no. 1, pp. 1–36, 2015.

[39]. Moler CB, Numerical computing with MATLAB. SIAM, 2004.

[40]. Tauzin G, Lupo U, Tunstall L, Pérez JB, Caorsi M, Medina-Mardones AM, Dassatti A, and Hess K, “giotto-tda:: A topological data analysis toolkit for machine learning and data exploration.” Journal of Machine Learning Research, vol. 22, pp. 39–1, 2021.

[41]. Bradski G and Kaehler A, Learning OpenCV: Computer vision with the OpenCV library. ” O’Reilly Media, Inc.”, 2008.

[42]. Jung A, “Imgaug documentation,” Readthedocs. io, Jun, vol. 25, 2019.

[43]. Clark A, “Pillow (pil fork) documentation,” Readthedocs. io,, 2015.

[44]. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V et al. , “Scikit-learn: Machine learning in python,” the Journal of machine Learning research, vol. 12, pp. 2825–2830, 2011.

[45]. Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, Burovski E, Peterson P, Weckesser W, Bright J et al. , "Scipy 1.0: fundamental algorithms for scientific computing in python," Nature methods, vol. 17, no. 3, pp. 261–272, 2020. [PubMed: 32015543]

[46]. Warden P and Situnayake D, Tinyml: Machine learning with tensor-flow lite on arduino and ultra-low-power microcontrollers. O'Reilly Media, 2019.

[47]. Krishnamoorthi R, "Quantizing deep convolutional networks for efficient inference: A whitepaper," arXiv preprint arXiv:1806.08342, 2018.

[48]. Zhu M and Gupta S, "To prune, or not to prune: exploring the efficacy of pruning for model compression," International Conference on Learning Representations, 2018.

[49]. Han S, Mao H, and Dally WJ, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," International Conference on Learning Representations (ICLR), 2016.

[50]. Coelho CN, Kuusela A, Li S, Zhuang H, Ngadiuba J, Aarrestad TK, Loncar V, Pierini M, Pol AA, and Summers S, "Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors," Nature Machine Intelligence, vol. 3, no. 8, pp. 675–686, 2021.

[51]. Siddegowda S, Fournarakis M, Nagel M, Blankevoort T, Patel C, and Khobare A, "Neural network quantization with ai model efficiency toolkit (aimet)," arXiv preprint arXiv:2201.08442, 2022.

[52]. Geiger L and Team P, "Larq: An open-source library for training binarized neural networks," Journal of Open Source Software, vol. 5, no. 45, p. 1746, 2020.

[53]. Bannink T, Hillier A, Geiger L, de Bruin T, Overweel L, Neeven J, and Helwegen K, "Larq compute engine: Design, benchmark and deploy state-of-the-art binarized neural networks," Proceedings of Machine Learning and Systems, vol. 3, pp. 680–695, 2021.

[54]. Microsoft, "microsoft/nni: An open source automl toolkit for automate machine learning lifecycle, including feature engineering, neural architecture search, model compression and hyper-parameter tuning." [Online]. Available: https://github.com/microsoft/nni

[55]. Capotondi A, Rusci M, Fariselli M, and Benini L, "Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices," IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 67, no. 5, pp. 871–875, 2020.

[56]. Gopinath S, Ghanathe N, Seshadri V, and Sharma R, "Compiling kb-sized machine learning models to tiny iot devices," in Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2019, pp. 79–95.

[57]. Kumar A, Goyal S, and Varma M, "Resource-efficient machine learning in 2 kb ram for the internet of things," in International Conference on Machine Learning. PMLR, 2017, pp. 1935–1944.

[58]. Gupta C, Suggala AS, Goyal A, Simhadri HV, Paranjape B, Kumar A, Goyal S, Udupa R, Varma M, and Jain P, "Protonn: Compressed and accurate knn for resource-scarce devices," in International Conference on Machine Learning. PMLR, 2017, pp. 1331–1340.

[59]. Kusupati A, Singh M, Bhatia K, Kumar A, Jain P, and Varma M, "Fastgrnn: a fast, accurate, stable and tiny kilobyte sized gated recurrent neural network," in Proceedings of the 32nd International Conference on Neural Information Processing Systems, 2018, pp. 9031–9042.

[60]. Saha O, Kusupati A, Simhadri HV, Varma M, and Jain P, "Rnnpool: Efficient non-linear pooling for ram constrained inference," Advances in Neural Information Processing Systems, vol. 33, 2020.

[61]. Denton EL, Zaremba W, Bruna J, LeCun Y, and Fergus R, "Exploiting linear structure within convolutional networks for efficient evaluation," in Advances in Neural Information Processing Systems, 2014, pp. 1269–1277.

[62]. Lee DD and Seung HS, "Learning the parts of objects by non-negative matrix factorization," Nature, vol. 401, no. 6755, pp. 788–791, 1999. [PubMed: 10548103]

[63]. Comon P, "Independent component analysis, a new concept?" Signal Proceedings, vol. 36, no. 3, pp. 287–314, 1994.

[64]. Balakrishnama S and Ganapathiraju A, "Linear discriminant analysis-a brief tutorial," Institute for Signal and Information Processing, vol. 18, no. 1998, pp. 1–8, 1998.

[65]. Espadoto M, Martins RM, Kerren A, Hirata NS, and Telea AC, “Toward a quantitative survey of dimension reduction techniques,” IEEE Transactions on visualization and Computer graphics, vol. 27, no. 3, pp. 2153–2173, 2019.

[66]. McInnes L, Healy J, Saul N, and Großberger L, “Umap: Uniform manifold approximation and projection,” Journal of Open Source Software, vol. 3, no. 29, 2018.

[67]. Kullback S and Leibler RA, “On information and sufficiency,” The Annals of Mathematical Statistics, vol. 22, no. 1, pp. 79–86, 1951.

[68]. Roweis ST and Saul LK, “Nonlinear dimensionality reduction by locally linear embedding,” Science, vol. 290, no. 5500, pp. 2323–2326, 2000. [PubMed: 11125150]

[69]. Schölkopf B, Smola A, and Müller K-R, “Kernel principal component analysis,” in International Conference on Artificial Neural Networks. Springer, 1997, pp. 583–588.

[70]. Van der Maaten L and Hinton G, “Visualizing data using t-sne.” Journal of Machine Learning Research, vol. 9, no. 11, 2008.

[71]. Rumelhart DE, Hinton GE, and Williams RJ, “Learning internal representations by error propagation,” California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[72]. “eloquentarduino.” [Online]. Available: https://github.com/eloquentarduino

[73]. Guyon I, Gunn S, Nikravesh M, and Zadeh LA, Feature extraction: foundations and applications. Springer, 2008, vol. 207.

[74]. Guyon I and Elisseeff A, “An introduction to variable and feature selection,” Journal of Machine Learning Research, vol. 3, no. Mar, pp. 1157–1182, 2003.

[75]. Chandrashekar G and Sahin F, “A survey on feature selection methods,” Computer & Electrical Engineering, vol. 40, no. 1, pp. 16–28, 2014.

[76]. Han S, Pool J, Tran J, and Dally W, “Learning both weights and connections for efficient neural network,” Advances in Neural Information Processing Systems, vol. 28, 2015.

[77]. Gupta S, Agrawal A, Gopalakrishnan K, and Narayanan P, “Deep learning with limited numerical precision,” in International Conference on Machine Learning. PMLR, 2015, pp. 1737–1746.

[78]. Yao S, Zhao Y, Zhang A, Su L, and Abdelzaher T, “Deepiot: Compressing deep neural network structures for sensing systems with a compressor-critic framework,” in Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems, 2017, pp. 1–14.

[79]. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M et al., “{TensorFlow}: A system for {Large-Scale} machine learning,” in 12th USENIX symposium on operating systems design and implementation (OSDI 16), 2016, pp. 265–283.

[80]. Jacob B, Kligys S, Chen B, Zhu M, Tang M, Howard A, Adam H, and Kalenichenko D, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 2704–2713.

[81]. Novac P-E, Hacene GB, Pegatoquet A, Miramond B, and Gripon V, “Quantization and deployment of deep neural networks on microcontrollers,” Sensors, vol. 21, no. 9, p. 2984, 2021. [PubMed: 33922868]

[82]. Wang K, Liu Z, Lin Y, Lin J, and Han S, “Haq: Hardware-aware automated quantization with mixed precision,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 8612–8620.

[83]. Rusci M, Capotondi A, and Benini L, “Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers,” Proceedings of Machine Learning and Systems, vol. 2, 2020.

[84]. Hubara I, Courbariaux M, Soudry D, El-Yaniv R, and Bengio Y, “Binarized neural networks,” Advances in Neural Information Processing Systems, vol. 29, 2016.

[85]. Dave S, Baghdadi R, Nowatzki T, Avancha S, Shrivastava A, and Li B, “Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights,” Proceedings of the IEEE, vol. 109, no. 10, pp. 1706–1752, 2021.

[86]. Fedorov I, Adams RP, Mattina M, and Whatmough PN, “Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers,” Advances in Neural Information Processing Systems, vol. 32, 2019.

[87]. Liu Z, Sun M, Zhou T, Huang G, and Darrell T, “Rethinking the value of network pruning,” in International Conference on Learning Representations, 2018.

[88]. Yu J, Lukefahr A, Palframan D, Dasika G, Das R, and Mahlke S, “Scalpel: Customizing dnn pruning to the underlying hardware parallelism,” in 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE Computer Society, 2017, pp. 548–560.

[89]. Liberis E and Lane ND, “Differentiable network pruning to enable smart applications,” Fourth UK Mobile, Wearable and Ubiquitous Systems Research Symposium, 2022.

[90]. Thakker U, Fedorov I, Zhou C, Gope D, Mattina M, Dasika G, and Beu J, “Compressing rnns to kilobyte budget for iot devices using kronecker products,” ACM Journal on Emerging Technologies in Computing Systems (JETC), vol. 17, no. 4, pp. 1–18, 2021.

[91]. Thakker U, Whatmough P, Liu Z, Mattina M, and Beu J, “Doping: A technique for extreme compression of lstm models using sparse structured additive matrices,” Proceedings of Machine Learning and Systems, vol. 3, 2021.

[92]. Iandola FN, Han S, Moskewicz MW, Ashraf K, Dally WJ, and Keutzer K, “Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size,” arXiv preprint arXiv:1602.07360, 2016.

[93]. Krizhevsky A, Sutskever I, and Hinton GE, “Imagenet classification with deep convolutional neural networks,” Advances in Neural Information Processing Systems, vol. 25, pp. 1097–1105, 2012.

[94]. Sandler M, Howard A, Zhu M, Zhmoginov A, and Chen L-C, “Mobilenetv2: Inverted residuals and linear bottlenecks,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 4510–4520.

[95]. Zhang X, Zhou X, Lin M, and Sun J, “Shufflenet: An extremely efficient convolutional neural network for mobile devices,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 6848–6856.

[96]. Huang G, Liu Z, Van Der Maaten L, and Weinberger KQ, “Densely connected convolutional networks,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 4700–4708.

[97]. Tan M, Pang R, and Le QV, “Efficientdet: Scalable and efficient object detection,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2020, pp. 10 781–10 790.

[98]. Wang C-Y, Bochkovskiy A, and Liao H-YM, “Scaled-yolov4: Scaling cross stage partial network,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 13 029–13 038.

[99]. Bochkovskiy A, Wang C-Y, and Liao H-YM, “Yolov4: Optimal speed and accuracy of object detection,” arXiv preprint arXiv:2004.10934, 2020.

[100]. Wang C-Y, Liao H-YM, Wu Y-H, Chen P-Y, Hsieh J-W, and Yeh I-H, “Cspnet: A new backbone that can enhance learning capability of cnn,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition WKSH, 2020, pp. 390–391.

[101]. Huang R, Pedoeem J, and Chen C, “Yolo-lite: a real-time object detection algorithm optimized for non-gpu computers,” in 2018 IEEE International Conference on Big Data (Big Data). IEEE, 2018, pp. 2503–2510.

[102]. Redmon J, Divvala S, Girshick R, and Farhadi A, “You only look once: Unified, real-time object detection,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 779–788.

[103]. Hochreiter S and Schmidhuber J, “Long short-term memory,” Neural computation, vol. 9, no. 8, pp. 1735–1780, 1997. [PubMed: 9377276]

[104]. Cho K, van Merriënboer B, Bahdanau D, and Bengio Y, “On the properties of neural machine translation: Encoder–decoder approaches,” in Proceedings of SSST-8, Eighth WKSH on Syntax, Semantics and Structure in Statistical Translation, 2014, pp. 103–111.

[105]. Arjovsky M, Shah A, and Bengio Y, “Unitary evolution recurrent neural networks,” in International Conference on Machine Learning. PMLR, 2016, pp. 1120–1128.

[106]. Jose C, Cissé M, and Fleuret F, “Kronecker recurrent units,” in International Conference on Machine Learning. PMLR, 2018, pp. 2380–2389.

[107]. Voelker A, Kajić I, and Eliasmith C, “Legendre memory units: Continuous-time representation in recurrent neural networks,” Advances in Neural Information Processing Systems, vol. 32, pp. 15 570–15 579, 2019.

[108]. van den Oord A, Dieleman S, Zen H, Simonyan K, Vinyals O, Graves A, Kalchbrenner N, Senior A, and Kavukcuoglu K, “Wavenet: A generative model for raw audio,” in 9th ISCA WKSH on Speech Synthesis WKSH (SSW 9), 2016.

[109]. Lea C, Vidal R, Reiter A, and Hager GD, “Temporal convolutional networks: A unified approach to action segmentation,” in European Conference on Computer Vision. Springer, 2016, pp. 47–54.

[110]. Ordóñez FJ and Roggen D, “Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition,” Sensors, vol. 16, no. 1, p. 115, 2016. [PubMed: 26797612]

[111]. Donahue J, Anne Hendricks L, Guadarrama S, Rohrbach M, Venu-gopalan S, Saenko K, and Darrell T, “Long-term recurrent convolutional networks for visual recognition and description,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 2625–2634.

[112]. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones Ł, Gomez AN, Kaiser E, and Polosukhin I, “Attention is all you need,” in Advances in Neural Information Processing Systems, 2017, pp. 5998–6008.

[113]. Wong A, Famouri M, Pavlova M, and Surana S, “Tinyspeech: Attention condensers for deep speech recognition neural networks on edge devices,” New in ML Workshop, NeurIPS, 2020.

[114]. Wong A, Famouri M, and Shafiee MJ, “Attendnets: Tiny deep image recognition neural networks for the edge via visual attention condensers,” 6th WKSH on Energy Efficient Machine Learning and Cognitive Computer (EMC2 2020), 2020.

[115]. Wen X, Famouri M, Hryniowski A, and Wong A, “Attendseg: A tiny attention condenser neural network for semantic segmentation on the edge,” arXiv preprint arXiv:2104.14623, 2021.

[116]. Wu Z, Liu Z, Lin J, Lin Y, and Han S, “Lite transformer with long-short range attention,” in International Conference on Learning Representations, 2019.

[117]. Mehta S and Rastegari M, “Mobilevit: Light-weight, general-purpose, and mobile-friendly vision transformer,” International Conference on Learning Representations (ICLR), 2022.

[118]. Jiao X, Yin Y, Shang L, Jiang X, Chen X, Li L, Wang F, and Liu Q, “Tinybert: Distilling bert for natural language understanding,” in Proceedings of the 2020 Conference on Empirical Methods in Natural Language Proceedings: Findings, 2020, pp. 4163–4174.

[119]. Burrello A, Scherer M, Zanghieri M, Conti F, and Benini L, “A microcontroller is all you need: Enabling transformer execution on low-power iot endnodes,” in 2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS). IEEE, 2021, pp. 1–6.

[120]. Abbasi S, Famouri M, Shafiee MJ, and Wong A, “Outliernets: Highly compact deep autoencoder network architectures for on-device acoustic anomaly detection,” Sensors (Basel, Switzerland), vol. 21, no. 14, p. 4805, 2021. [PubMed: 34300545]

[121]. Lin J, Chen W-M, Cai H, Gan C, and Han S, “Mcunetv2: Memory-efficient patch-based inference for tiny deep learning,” in Annual Conference on Neural Information Processing Systems (NeurIPS), 2021.

[122]. Liberis E, Dudziak Ł, and Lane ND, “μnas: Constrained neural architecture search for microcontrollers,” in Proceedings of the 1st WKSH on Machine Learning and Systems, 2021, pp. 70–79.

[123]. Sandha SS, Aggarwal M, Saha SS, and Srivastava M, “Enabling hyperparameter tuning of machine learning classifiers in production,” in 2021 IEEE Third International Conference on Cognitive Machine Intelligence (CogMI). IEEE, 2021, pp. 1–10.

[124]. Mendis HR, Kang C-K, and Hsiu P.-c., “Intermittent-aware neural architecture search,” ACM Transactions on Embedded Computing Systems (TECS), vol. 20, no. 5s, pp. 1–27, 2021.

[125]. Ren P, Xiao Y, Chang X, Huang P-Y, Li Z, Chen X, and Wang X, “A comprehensive survey of neural architecture search: Challenges and solutions,” ACM Computing Surveys (CSUR), vol. 54, no. 4, pp. 1–34, 2021.

[126]. Zoph B and Le QV, “Neural architecture search with reinforcement learning,” International Conference on Learning Representations (ICLR), 2017.

[127]. Baker B, Gupta O, Naik N, and Raskar R, “Designing neural network architectures using reinforcement learning,” International Conference on Learning Representations (ICLR), 2017.

[128]. Zoph B, Vasudevan V, Shlens J, and Le QV, “Learning transferable architectures for scalable image recognition,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 8697–8710.

[129]. Cai H, Zhu L, and Han S, “Proxylessnas: Direct neural architecture search on target task and hardware,” in International Conference on Learning Representations, 2018.

[130]. Chen T, Moreau T, Jiang Z, Zheng L, Yan E, Shen H, Cowan M, Wang L, Hu Y, Ceze L 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.

[131]. Li Z, Xi T, Deng J, Zhang G, Wen S, and He R, “Gp-nas: Gaussian process based neural architecture search,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 11933–11942.

[132]. Tan M, Chen B, Pang R, Vasudevan V, Sandler M, Howard A, and Le QV, “Mnasnet: Platform-aware neural architecture search for mobile,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 2820–2828.

[133]. Howard A, Sandler M, Chu G, Chen L-C, Chen B, Tan M, Wang W, Zhu Y, Pang R, Vasudevan V et al., “Searching for mobilenetv3,” in Proceedings of the IEEE/CVF International Conference on Computer Vision, 2019, pp. 1314–1324.

[134]. Elsken T, Metzen JH, and Hutter F, “Neural architecture search: A survey,” Journal of Machine Learning Research, vol. 20, no. 1, pp. 1997–2017, 2019.

[135]. Li G, Mandal SK, Ogras UY, and Marculescu R, “Flash: F ast neura l a rchitecture s earch with h ardware optimization,” ACM Transactions on Embedded Computing Systems (TECS), vol. 20, no. 5s, pp. 1–26, 2021.

[136]. Watkins CJ and Dayan P, “Q-learning,” Machine learning, vol. 8, no. 3, pp. 279–292, 1992.

[137]. Silver D, Lever G, Heess N, Degris T, Wierstra D, and Riedmiller M, “Deterministic policy gradient algorithms,” in International conference on machine learning. PMLR, 2014, pp. 387–395.

[138]. Sutton RS and Barto AG, Reinforcement learning: An introduction. MIT press, 2018.

[139]. Liu H, Simonyan K, and Yang Y, “Darts: Differentiable architecture search,” in International Conference on Learning Representations, 2018.

[140]. Wu B, Dai X, Zhang P, Wang Y, Sun F, Wu Y, Tian Y, Vajda P, Jia Y, and Keutzer K, “Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 10734–10742.

[141]. Guo Z, Zhang X, Mu H, Heng W, Liu Z, Wei Y, and Sun J, “Single path one-shot neural architecture search with uniform sampling,” in European Conference on Computer Vision. Springer, 2020, pp. 544–560.

[142]. Elsken T, Metzen JH, and Hutter F, “Efficient multi-objective neural architecture search via lamarckian evolution,” in International Conference on Learning Representations, 2018.

[143]. Pham H, Guan M, Zoph B, Le Q, and Dean J, “Efficient neural architecture search via parameters sharing,” in International conference on machine learning. PMLR, 2018, pp. 4095–4104.

[144]. Stamoulis D, Ding R, Wang D, Lymberopoulos D, Priyantha B, Liu J, and Marculescu D, “Single-path nas: Designing hardware-efficient convnets in less than 4 hours,” in Joint European Conference on Machine Learning and Knowledge Discovery in Databases. Springer, 2019, pp. 481–497.

[145]. Real E, Aggarwal A, Huang Y, and Le QV, “Regularized evolution for image classifier architecture search,” in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, no. 01, 2019, pp. 4780–4789.

[146]. Sandha SS, Aggarwal M, Fedorov I, and Srivastava M, “Mango: A python library for parallel hyperparameter tuning,” in ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2020, pp. 3987–3991.

[147]. Golovin D, Solnik B, Moitra S, Kochanski G, Karro J, and Sculley D, “Google vizier: A service for black-box optimization,” in Proceedings of the 23rd ACM SIGKDD International Conference on knowledge discovery and data mining, 2017, pp. 1487–1495.

[148]. Daulton S, Eriksson D, Balandat M, and Bakshy E, “Multi-objective bayesian optimization over high-dimensional search spaces,” in The 38th Conference on Uncertainty in Artificial Intelligence, 2022.

[149]. Dudziak L, Chau T, Abdelfattah M, Lee R, Kim H, and Lane N, “Brp-nas: Prediction-based nas using gcns,” Advances in Neural Information Processing Systems, vol. 33, pp. 10480–10490, 2020.

[150]. Wen W, Liu H, Chen Y, Li H, Bender G, and Kindermans P-J, “Neural predictor for neural architecture search,” in European Conference on Computer Vision. Springer, 2020, pp. 660–676.

[151]. Mellor J, Turner J, Storkey A, and Crowley EJ, “Neural architecture search without training,” in International Conference on Machine Learning. PMLR, 2021, pp. 7588–7598.

[152]. Abdelfattah MS, Mehrotra A, Dudziak Ł, and Lane ND, “Zero-cost proxies for lightweight nas,” in International Conference on Learning Representations, 2020.

[153]. Lee N, Ajanthan T, and Torr P, “Snip: Single-shot network pruning based on connection sensitivity,” in International Conference on Learning Representations, 2018.

[154]. Wang C, Zhang G, and Grosse R, “Picking winning tickets before training by preserving gradient flow,” in International Conference on Learning Representations, 2019.

[155]. Tanaka H, Kunin D, Yamins DL, and Ganguli S, “Pruning neural networks without any data by iteratively conserving synaptic flow,” Advances in Neural Information Processing Systems, vol. 33, pp. 6377–6389, 2020.

[156]. Chen W, Gong X, and Wang Z, “Neural architecture search on imagenet in four gpu hours: A theoretically inspired perspective,” in International Conference on Learning Representations, 2020.

[157]. Gielda M, “Renode: a flexible, open-source simulation framework for building scalable, well-tested risc-v systems,” 7th RISC-V Workshop Proceedings, 2017.

[158]. Zhang LL, Han S, Wei J, Zheng N, Cao T, Yang Y, and Liu Y, “nn-meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices,” in Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services, 2021, pp. 81–93.

[159]. Bulo SR, Porzi L, and Kontschieder P, “In-place activated batchnorm for memory-optimized training of dnns,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 5639–5647.

[160]. Xue J, Loop tiling for parallelism. Springer Science & Business Media, 2000, vol. 575.

[161]. Carr S, McKinley KS, and Tseng C-W, “Compiler optimizations for improving data locality,” ACM SIGPLAN Notices, vol. 29, no. 11, pp. 252–262, 1994.

[162]. Dave S, Kim Y, Avancha S, Lee K, and Shrivastava A, “Dmazerunner: Executing perfectly nested loops on dataflow accelerators,” ACM Transactions on Embedded Computing Systems (TECS), vol. 18, no. 5s, pp. 1–27, 2019. [PubMed: 34084098]

[163]. Yu J, Lukefahr A, Das R, and Mahlke S, “Tf-net: Deploying subbyte deep neural networks on microcontrollers,” ACM Transactions on Embedded Computing Systems (TECS), vol. 18, no. 5s, pp. 1–21, 2019. [PubMed: 34084098]

[164]. Lai L, Suda N, and Chandra V, “Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus,” arXiv preprint arXiv:1801.06601, 2018.

[165]. Burrello A, Garofalo A, Bruschi N, Tagliavini G, Rossi D, and Conti F, “Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus,” IEEE Transactions on Computers, vol. 70, no. 8, pp. 1253–1268, 2021.

[166]. Garofalo A, Rusci M, Conti F, Rossi D, and Benini L, “Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors,” Philosophical Transactions of the Royal Society A, vol. 378, no. 2164, p. 20190155, 2020.

[167]. David R, Duke J, Jain A, Janapa Reddi V, Jeffries N, Li J, Kreeger N, Nappier I, Natraj M, Wang T et al. , "Tensorflow lite micro: Embedded machine learning for tinyml systems," Proceedings of Machine Learning and Systems, vol. 3, 2021.

[168]. Tan N, "utensor/utensor: Tinyml ai inference library" [Online]. Available: https://github.com/uTensor/uTensor

[169]. Dennis DK, Acar DAE, Mandikal V, Sadasivan VS, Saligrama V, Simhadri HV, and Jain P, "Shallow rnn: Accurate time-series classification on resource constrained devices," in Advances in Neural Information Processing Systems, vol. 32, 2019.

[170]. Dennis DK, Pabbaraju C, Simhadri HV, and Jain P, "Multiple instance learning for efficient sequential data classification on resource-constrained devices," in Proceedings of the 32nd International Conference on Neural Information Processing Systems, 2018, pp. 10 976–10 987.

[171]. Goyal S, Raghunathan A, Jain M, Simhadri HV, and Jain P, "Drocc: Deep robust one-class classification," in International Conference on Machine Learning. PMLR, 2020, pp. 3711–3721.

[172]. "X-cube-ai - ai expansion pack for stm32cubemx - stmicroelectronics." [Online]. Available: https://www.st.com/en/embedded-software/x-cube-ai.html

[173]. "Cartesiam, nanoedge ai library," Dec 2021. [Online]. Available: https://cartesiam.ai/

[174]. Morawiec D, "sklearn-porter." [Online]. Available: https://github.com/nok/sklearn-porter

[175]. da Silva LT, Souza VM, and Batista GE, "Embml tool: Supporting the use of supervised learning algorithms in low-cost embedded systems," in 2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI). IEEE, 2019, pp. 1633–1637.

[176]. Wang X, Magno M, Cavigelli L, and Benini L, "Fann-on-mcu: An open-source toolkit for energy-efficient neural network inference at the edge of the internet of things," IEEE Internet of Things Journal, vol. 7, no. 5, pp. 4403–4417, 2020.

[177]. "Ai-driven deep neural network optimizer." [Online]. Available: https://www.deeplite.ai/

[178]. "Edge ai / tinyml: Saas: Deep learning." [Online]. Available: https://www.imagimob.com/

[179]. "Neuton.ai - no-code artificial intelligence for all." [Online]. Available: https://neuton.ai/

[180]. "Reality ai." [Online]. Available: https://reality.ai/

[181]. Whaley RC and Dongarra JJ, "Automatically tuned linear algebra software," in SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing. IEEE, 1998, pp. 38–38.

[182]. Bondhugula U, Hartono A, Ramanujam J, and Sadayappan P, "A practical automatic polyhedral parallelizer and locality optimizer," in Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2008, pp. 101–113.

[183]. Lee S and Nirjon S, "Learning in the wild: When, how, and what to learn for on-device dataset adaptation," in Proceedings of the 2nd International WKSH on Challenges in Artificial Intelligence and Machine Learning for Internet of Things, 2020, pp. 34–40.

[184]. Ren H, Anicic D, and Runkler TA, "Tinyol: Tinyml with onlinelearning on microcontrollers," in 2021 International Joint Conference on Neural Networks (IJCNN). IEEE, 2021, pp. 1–8.

[185]. Sudharsan B, Breslin JG, and Ali MI, "Ml-mcu: A framework to train ml classifiers on mcu-based iot edge devices," IEEE Internet of Things Journal, 2021.

[186]. Sudharsan B, Yadav P, Breslin JG, and Ali MI, "Train++: An incremental ml model training algorithm to create self-learning iot devices," in 2021 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/IOP/SCI). IEEE, 2021, pp. 97–106.

[187]. Sudharsan B, Breslin JG, and Ali MI, "Imbal-ol: Online machine learning from imbalanced data streams in real-world iot," in 2021 IEEE International Conference on Big Data (Big Data). IEEE, 2021, pp. 4974–4978.

[188]. Ravaglia L, Rusci M, Nadalini D, Capotondi A, Conti F, and Benini L, "A tinyml platform for on-device continual learning with quantized latent replays," IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 11, no. 4, pp. 789–802, 2021.

[189]. Osman A, Abid U, Gemma L, Perotto M, and Brunelli D, "Tinyml platforms benchmarking," in International Conference on Applications in Electronics Pervading Industry, Environment and Society. Springer, 2022, pp. 139–148.

[190]. Beutel DJ, Topal T, Mathur A, Qiu X, Parcollet T, de Gusmáo PP, and Lane ND, “Flower: A friendly federated learning research framework,” arXiv preprint arXiv:2007.14390, 2020.

[191]. Mathur A, Beutel DJ, de Gusmao PPB, Fernandez-Marques J, Topal T, Qiu X, Parcollet T, Gao Y, and Lane ND, “On-device federated learning with flower,” On-Device Intelligence Workshop at MLSys, 2021.

[192]. Imteaj A and Amini MH, “Fedparl: Client activity and resource-oriented lightweight federated learning model for resource-constrained heterogeneous iot environment,” Frontiers in Communications and Networks, p. 10, 2021.

[193]. Nguyen TD, Marchal S, Miettinen M, Fereidooni H, Asokan N, and Sadeghi A-R, “Dïot: A federated self-learning anomaly detection system for iot,” in 2019 IEEE 39th International conference on distributed computing systems (ICDCS). IEEE, 2019, pp. 756–767.

[194]. Jiang Y, Wang S, Valls V, Ko BJ, Lee W-H, Leung KK, and Tassiulas L, “Model pruning enables efficient federated learning on edge devices,” IEEE Transactions on Neural Networks and Learning Systems, 2022.

[195]. Kopparapu K, Lin E, Breslin JG, and Sudharsan B, “Tinyfedtl: Federated transfer learning on ubiquitous tiny iot devices,” in 2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops). IEEE, 2022, pp. 79–81.

[196]. Pang J, Huang Y, Xie Z, Han Q, and Cai Z, “Realizing the heterogeneity: a self-organized federated learning framework for iot,” IEEE Internet of Things Journal, vol. 8, no. 5, pp. 3088–3098, 2020.

[197]. Wu Q, He K, and Chen X, “Personalized federated learning for intelligent iot applications: A cloud-edge based framework,” IEEE Open Journal of the Computer Society, vol. 1, pp. 35–44, 2020.

[198]. McMahan B, Moore E, Ramage D, Hampson S, and y Arcas BA, “Communication-efficient learning of deep networks from decentralized data,” in Artificial intelligence and statistics. PMLR, 2017, pp. 1273–1282.

[199]. Jha NK, Mittal S, and Mattela G, “The ramifications of making deep neural networks compact,” in 2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID). IEEE, 2019, pp. 215–220.

[200]. Paul AJ, Mohan P, and Sehgal S, “Rethinking generalization in american sign language prediction for edge devices with extremely low memory footprint,” in 2020 IEEE Recent Advances in Intelligent Computational Systems (RAICS). IEEE, 2020, pp. 147–152.

[201]. Mohan P, Paul AJ, and Chirania A, “A tiny cnn architecture for medical face mask detection for resource-constrained endpoints,” in Innovations in Electrical and Electronic Engineering. Springer, 2021, pp. 657–670.

[202]. Yoo J, Lee D, Son C, Jung S, Yoo B, Choi C, Han J-J, and Han B, “Rascanet: Learning tiny models by raster-scanning images,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 13 673–13 682.

[203]. Nasr M, Shokri R, and Houmansadr A, “Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning,” in 2019 IEEE symposium on security and privacy (SP). IEEE, 2019, pp. 739–753.

[204]. Jere MS, Farnan T, and Koushanfar F, “A taxonomy of attacks on federated learning,” IEEE Security & Privacy, vol. 19, no. 2, pp. 20–28, 2020.

[205]. Li KH, de Gusmão PPB, Beutel DJ, and Lane ND, “Secure aggregation for federated learning in flower,” in Proceedings of the 2nd ACM International Workshop on Distributed Machine Learning, 2021, pp. 8–14.

[206]. Rawat W and Wang Z, “Deep convolutional neural networks for image classification: A comprehensive review,” Neural computation, vol. 29, no. 9, pp. 2352–2449, 2017. [PubMed: 28599112]

[207]. Blouw P, Malik G, Morcos B, Voelker A, and Eliasmith C, “Hardware aware training for efficient keyword spotting on general purpose and specialized hardware,” in Research Symposium on Tiny Machine Learning, 2020.

[208]. Fedorov I, Stamenovic M, Jensen C, Yang L-C, Mandell A, Gan Y, Mattina M, and Whatmough PN, “Tinylstms: Efficient neural speech enhancement for hearing aids,” in Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH. ISCA, 2020, pp. 4054–4058.

[209]. Hardy E and Badets F, “An ultra-low power rnn classifier for always-on voice wake-up detection robust to real-world scenarios,” in Research Symposium on Tiny Machine Learning, 2020.

[210]. Mermelstein P, “Distance measures for speech recognition, psychological and instrumental,” Pattern Recognition and artificial intelligence, vol. 116, pp. 374–388, 1976.

[211]. Kayan H, Majib Y, Alsafery W, Barhamgi M, and Perera C, “Anomliot: An end to end re-configurable multi-protocol anomaly detection pipeline for internet of things,” Internet of Things, vol. 16, p. 100437, 2021.

[212]. Patil SG, Dennis DK, Pabbaraju C, Shaheer N, Simhadri HV, Seshadri V, Varma M, and Jain P, “Gesturepod: Enabling on-device gesture-based interaction for white cane users,” in Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology, 2019, pp. 403–415.

[213]. Bian S and Lukowicz P, “Capacitive sensing based on-board hand gesture recognition with tinyml,” in Adjunct Proceedings of the 2021 ACM International Joint Conference on Pervasive and Ubiquitous Computer and Proceedings of the 2021 ACM International Symposium on Wearable Computer, 2021, pp. 4–5.

[214]. T’Jonck K, Kancharla CR, Vankeirsbilck J, Hallez H, Boydens J, and Pang B, “Real-time activity tracking using tinyml to support elderly care,” in 2021 International Scientific Conference Electronics (ET). IEEE, 2021, pp. 1–6.

[215]. Zhou A, Muller R, and Rabaey J, “Memory-efficient, limb position-aware hand gesture recognition using hyperdimensional computing,” in Research Symposium on Tiny Machine Learning, 2020.

[216]. Elsts A and McConville R, “Are microcontrollers ready for deep learning-based human activity recognition?” Electronics, vol. 10, no. 21, p. 2640, 2021.

[217]. Coelho YL, dos Santos F. d. A. S., Frizera-Neto A, and Bastos-Filho TF, “A lightweight framework for human activity recognition on wearable devices,” IEEE Sensors Journal, vol. 21, no. 21, pp. 24 471–24 481, 2021.

[218]. Badino H, Yamamoto A, and Kanade T, “Visual odometry by multiframe feature integration,” in Proceedings of the IEEE International Conference on Computer Vision WKSH, 2013, pp. 222–229.

[219]. Loquercio A, Maqueda AI, Del-Blanco CR, and Scaramuzza D, “Dronet: Learning to fly by driving,” IEEE Robotics and Automation Letters, vol. 3, no. 2, pp. 1088–1095, 2018.

[220]. Palossi D, Loquercio A, Conti F, Flamand E, Scaramuzza D, and Benini L, “A 64-mw dnn-based visual navigation engine for autonomous nano-drones,” IEEE Internet of Things Journal, vol. 6, no. 5, pp. 8357–8371, 2019.

[221]. Nyamukuru MT and Odame KM, “Tiny eats: Eating detection on a microcontroller,” in 2020 IEEE Second WKSH on Machine Learning on Edge in Sensor Systems (SenSys-ML). IEEE, 2020, pp. 19–23.

[222]. John A, Cardiff B, and John D, “A 1d-cnn based deep learning technique for sleep apnea detection in iot sensors,” in 2021 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, 2021, pp. 1–5.

[223]. Petrović N and Kocić, “Iot for covid-19 indoor spread prevention: Cough detection, air quality control and contact tracing,” in 2021 IEEE 32nd International Conference on Microelectronics (MIEL). IEEE, 2021, pp. 297–300.

[224]. Zemlyanikin M, Smorkalov A, Khanova T, Petrovicheva A, and Serebryakov G, “512kib ram is enough! live camera face recognition dnn on mcu,” in Proceedings of the IEEE/CVF International Conference on Computer Vision WKSH, 2019, pp. 0–0.

[225]. Zhao X, Liang X, Zhao C, Tang M, and Wang J, “Real-time multiscale face detector on embedded devices,” Sensors, vol. 19, no. 9, p. 2158, 2019. [PubMed: 31075955]

[226]. Wang M and Deng W, “Deep face recognition: A survey,” Neurocomputing, vol. 429, pp. 215–244, 2021.

[227]. Saha SS, Sandha SS, and Srivastava M, “Deep convolutional bidirectional lstm for complex activity recognition with missing data,” in Human Activity Recognition Challenge. Springer, 2020, pp. 39–53.

[228]. Hammerla NY, Halloran S, and Plötz T, “Deep, convolutional, and recurrent models for human activity recognition using wearables,” in Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, 2016, pp. 1533–1540.

[229]. Sainath T and Parada C, “Convolutional neural networks for small-footprint keyword spotting,” Sixteenth Annual Conference of the International Speech Communication Association, 2015.

[230]. Chen Y, Yang T-J, Emer J, and Sze V, “Understanding the limitations of existing energy-efficient design approaches for deep neural networks,” Energy, vol. 2, no. L1, p. L3, 2018.

[231]. Chen C, Zhao P, Lu CX, Wang W, Markham A, and Trigoni N, “Deep-learning-based pedestrian inertial navigation: Methods, data set, and on-device inference,” IEEE Internet of Things Journal, vol. 7, no. 5, pp. 4431–4441, 2020.

[232]. Herath S, Yan H, and Furukawa Y, “Ronin: Robust neural inertial navigation in the wild: Benchmark, evaluations, & new methods,” in 2020 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2020, pp. 3146–3152.

[233]. Deittert M, Richards A, Toomer CA, and Pipe A, “Engineless unmanned aerial vehicle propulsion by dynamic soaring,” Journal of guidance, control, and dynamics, vol. 32, no. 5, pp. 1446–1457, 2009.

[234]. Rohrbach M, Amin S, Andriluka M, and Schiele B, “A database for fine grained activity detection of cooking activities,” in 2012 IEEE Conference on Computer Vision and Pattern Recognition. IEEE, 2012, pp. 1194–1201.

[235]. Ma M, Gao J, Feng L, and Stankovic J, “Stlnet: Signal temporal logic enforced multivariate recurrent neural networks,” in Advances in Neural Information Processing Systems, Larochelle H, Ranzato M, Hadsell R, Balcan MF, and Lin H, Eds., vol. 33, 2020, pp. 14 604–14 614.

[236]. Xing T, Garcia L, Vilamala MR, Cerutti F, Kaplan L, Preece A, and Srivastava M, “Neuroplex: learning to detect complex events in sensor networks through knowledge injection,” in Proceedings of the 18th Conference on Embedded Networked Sensor Systems, 2020, pp. 489–502.

[237]. Mao J, Gan C, Kohli P, Tenenbaum JB, and Wu J, “The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision,” in International Conference on Learning Representations, 2018.

[238]. Greydanus S, Dzamba M, and Yosinski J, “Hamiltonian neural networks,” Advances in Neural Information Processing Systems, vol. 32, pp. 15 379–15 389, 2019.

[239]. Cranmer M, Greydanus S, Hoyer S, Battaglia P, Spergel D, and Ho S, “Lagrangian neural networks,” in ICLR 2020 WKSH on Integration of Deep Neural Models and Differential Equations, 2020.

[240]. Yao S, Piao A, Jiang W, Zhao Y, Shao H, Liu S, Liu D, Li J, Wang T, Hu S et al., “Stfnets: Learning sensing signals from the time-frequency perspective with short-time fourier neural networks,” in The World Wide Web Conference, 2019, pp. 2192–2202.

[241]. Li S, Chowdhury RR, Shang J, Gupta RK, and Hong D, “Units: Short-time fourier inspired neural networks for sensory time series classification,” in Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems, 2021, pp. 234–247.

[242]. Sandha SS, Noor J, Anwar FM, and Srivastava M, “Time awareness in deep learning-based multimodal fusion across smartphone platforms,” in 2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI). IEEE, 2020, pp. 149–156.

[243]. Stisen A, Blunck H, Bhattacharya S, Prentow TS, Kjærgaard MB, Dey A, Sonne T, and Jensen MM, “Smart devices are different: Assessing and mitigating mobile sensing heterogeneities for activity recognition,” in Proceedings of the 13th ACM Conference on embedded networked Sensor Systems, 2015, pp. 127–140.

[244]. Hossain T, Ahad M, Rahman A, and Inoue S, “A method for sensor-based activity recognition in missing data scenario,” Sensors, vol. 20, no. 14, p. 3811, 2020. [PubMed: 32650486]

[245]. Sandha SS, Anwar FM, Noor J, and Srivastava M, “Exploiting smartphone peripherals for precise time synchronization,” in 2019 IEEE Global Conference on Signal and Information Processing (GlobalSIP). IEEE, 2019, pp. 1–6.

[246]. Dhar S, Guo J, Liu J, Tripathi S, Kurup U, and Shah M, “A survey of on-device machine learning: An algorithms and learning theory perspective,” ACM Transactions on Internet of Things, vol. 2, no. 3, pp. 1–49, 2021.

[247]. Wu Y, Wang Z, Shi Y, and Hu J, “Enabling on-device cnn training by self-supervised instance filtering and error map pruning,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 11, pp. 3445–3457, 2020.

[248]. Bengio Y, Courville A, and Vincent P, “Representation learning: A review and new perspectives,” IEEE Transactions on Pattern analysis and machine intelligence, vol. 35, no. 8, pp. 1798–1828, 2013. [PubMed: 23787338]

[249]. Ganin Y, Ustinova E, Ajakan H, Germain P, Larochelle H, Laviolette F, Marchand M, and Lempitsky V, “Domain-adversarial training of neural networks,” Journal of machine Learning research, vol. 17, no. 1, pp. 2096–2030, 2016.

[250]. Srivastava M, Nushi B, Kamar E, Shah S, and Horvitz E, “An empirical analysis of backward compatibility in machine learning systems,” in Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2020, pp. 3272–3280.

[251]. Yan S, Xiong Y, Kundu K, Yang S, Deng S, Wang M, Xia W, and Soatto S, “Positive-congruent training: Towards regression-free model updates,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 14 299–14 308.

[252]. Bansal G, Nushi B, Kamar E, Weld DS, Lasecki WS, and Horvitz E, “Updates in human-ai teams: Understanding and addressing the performance/compatibility tradeoff,” in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, no. 01, 2019, pp. 2429–2437.

[253]. Jagielski M, Carlini N, Berthelot D, Kurakin A, and Papernot N, “High accuracy and high fidelity extraction of neural networks,” in 29th {USENIX} Security Symposium ({USENIX} Security 20), 2020, pp. 1345–1362.

[254]. Shen Y, Xiong Y, Xia W, and Soatto S, “Towards backward-compatible representation learning,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 6368–6377.

[255]. Lin J, Gan C, and Han S, “Defensive quantization: When efficiency meets robustness,” in International Conference on Learning Representations, 2018.

[256]. Gui S, Wang HN, Yang H, Yu C, Wang Z, and Liu J, “Model compression with adversarial robustness: A unified optimization framework,” Advances in Neural Information Processing Systems, vol. 32, pp. 1285–1296, 2019.

[257]. Ye S, Xu K, Liu S, Cheng H, Lambrechts J-H, Zhang H, Zhou A, Ma K, Wang Y, and Lin X, “Adversarial robustness vs. model compression, or both?” in Proceedings of the IEEE/CVF International Conference on Computer Vision, 2019, pp. 111–120.

[258]. Hassija V, Chamola V, Saxena V, Jain D, Goyal P, and Sikdar B, “A survey on iot security: application areas, security threats, and solution architectures,” IEEE Access, vol. 7, pp. 82 721–82743, 2019.

[259]. Guo M, Yang Y, Xu R, Liu Z, and Lin D, “When nas meets robustness: In search of robust architectures against adversarial attacks,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 631–640.

[260]. Xu X, Ding Y, Hu SX, Niemier M, Cong J, Hu Y, and Shi Y, “Scaling for edge inference of deep neural networks,” Nature Electronics, vol. 1, no. 4, pp. 216–222, 2018.

[261]. Liu Y, Liu S, Wang Y, Lombardi F, and Han J, “A survey of stochastic computing neural networks for machine learning applications,” IEEE Transactions on Neural Networks and Learning Systems, 2020.

[262]. Jiang W, Zhang X, Sha EH-M, Yang L, Zhuge Q, Shi Y, and Hu J, “Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search,” in Proceedings of the 56th Annual Design Automation Conference 2019, 2019, pp. 1–6.

[263]. Jiang W, Yang L, Sha EH-M, Zhuge Q, Gu S, Dasgupta S, Shi Y, and Hu J, “Hardware/software co-exploration of neural architectures,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 12, pp. 4805–4815, 2020.

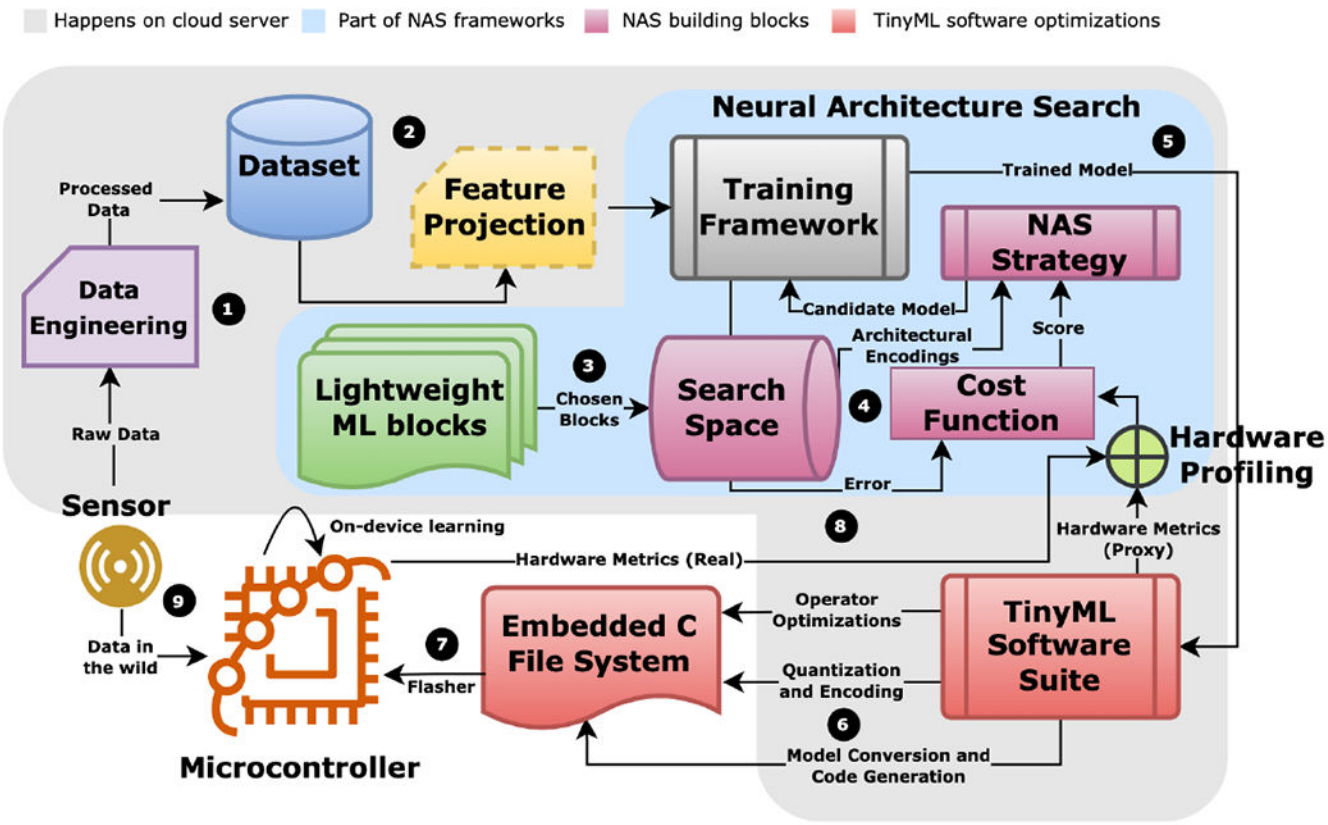


Fig. 1.

Closed-loop workflow of porting machine learning models onto microcontrollers. Step (3) to Step (8) are repeated until desired performance is achieved. (1) Data engineering performs acquisition, analytics and storage of raw sensor streams (Section III). (2) Optional feature projection directly reduces dimensionality of input data (Section IV). (3) Models are chosen from a lightweight ML zoo based on the application and hardware specifications (Section VI and Section X). (4) Neural architecture search strategy builds candidate models from the search space for training and evaluates the model based on cost function (Section VII). (5) Trained candidate model is ported to a TinyML software suite. (6) The TinyML software suite performs inference engine optimizations, deep compression and code generation. It also provides approximate hardware metrics (e.g., SRAM, Flash and latency) (Section V, Section VII, and Section VIII). (7) The embedded C file system is ported onto the microcontroller via command line interface. (8) The microcontroller optionally reports real runtime hardware metrics back to the neural architecture search strategy (Section VII). (9) On-device training or federated learning are used occasionally to account for shifts in incoming data distribution (Section IX).

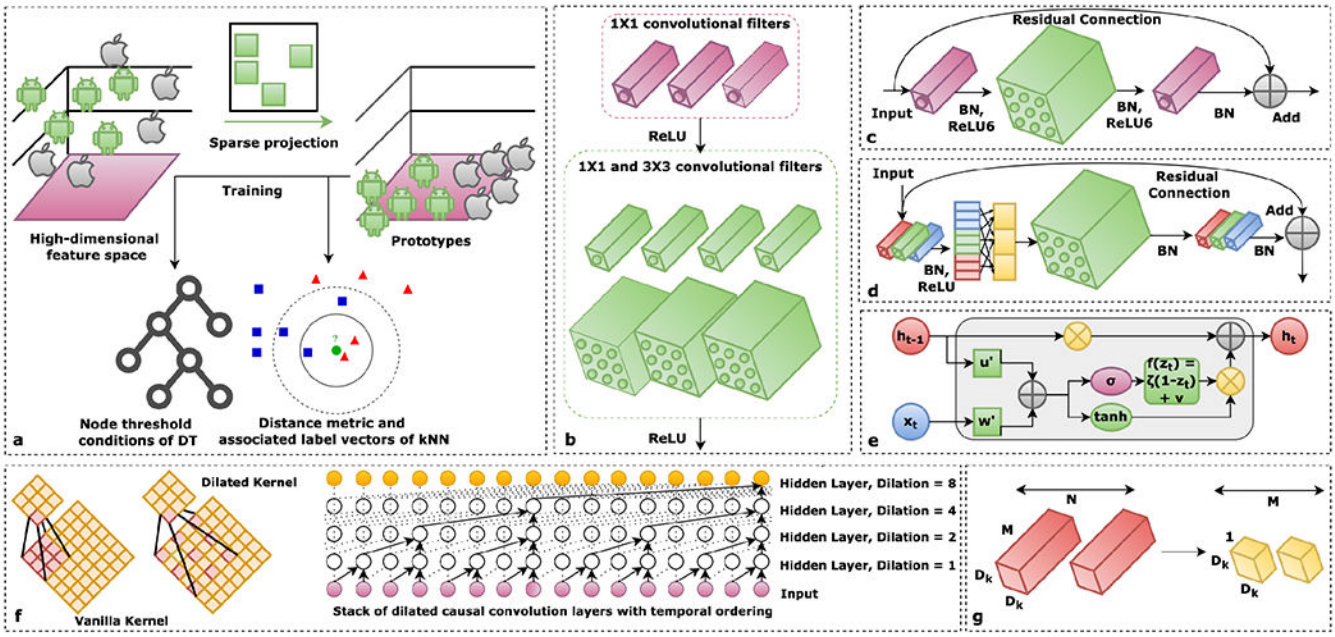


Fig. 2.

Example of lightweight machine learning blocks. (a) Sparse projection onto low-dimensional linear manifold yields lightweight decision trees and k-nearest neighbor classifiers. (b) Fire module containing bottleneck (pointwise) and excitation (pointwise and depthwise) convolutional layers. (c) The inverted residual connection between squeeze layers instead of excitation layers reduces memory and compute. (d) Group convolution with channel shuffle improves cross-channel relations. (e) Adding a gated residual connection and enforcing RNN matrices to be low rank, sparse, and quantized yields stable and lightweight RNN. (f) Temporal convolutional networks extract spatio-temporal representations using causal and dilated convolution kernels. (g) Depthwise separable convolution yields 7-9× memory savings over vanilla convolution kernel (figure adapted from [2]).

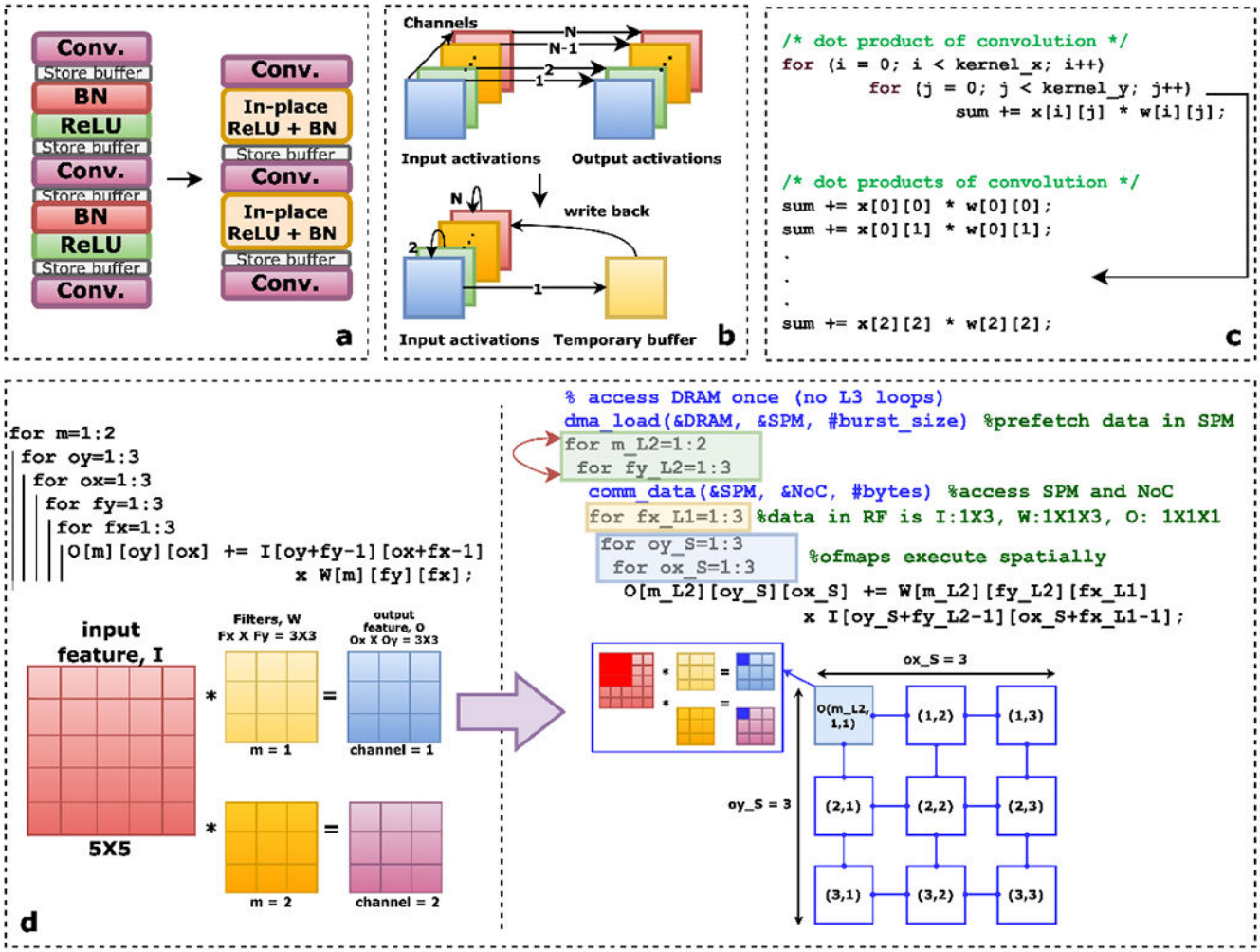


Fig. 3.

Example operator optimizations performed by TinyML software suites. (a) Use of fused and in-place activated operators reduce memory access cost and improves inference speed [158] [159]. (b) Converting depthwise convolution to in-place depthwise convolution reduces peak memory usage by 1.6×, by allowing first channel output activation (stored in a buffer) to overwrite the previous channel's inpur activation until written back to the last channel's input activation [31]. (c) Loop unrolling eliminates branch instruction overhead [31]. (d) Loop tiling encourages resuse of array elements within each tile by partitioning the loop's iterative space into blocks [160], while loop reordering (with tiling) improves spatiotemporal execution and locality of reference within device memory constraints [161] [162].

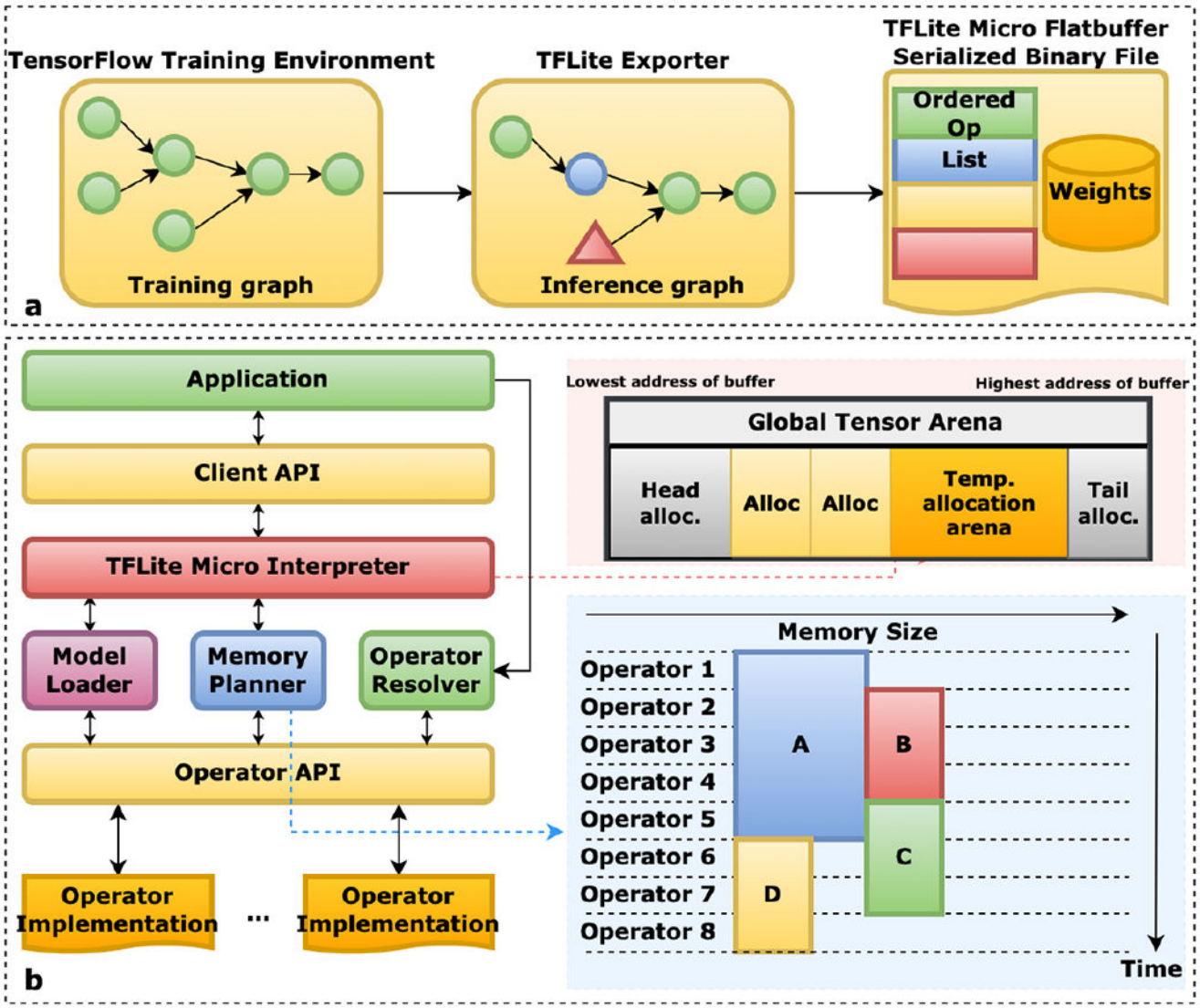


Fig. 4.

Operation of TensorFlow Lite Micro, an interpreter-based inference engine. (a) The training graph is frozen, optimized and converted to a flatbuffer serialized model schema, suitable for deployment in embedded devices. (b) The TFLM runtime API preallocates a portion of memory in the SRAM (called arena) and performs bin-packing during runtime to optimize memory usage (figure adapted from [167]).

TABLE I

COMPARISON OF HARDWARE FOR DOING MACHINE LEARNING ON CLOUD SERVERS, MOBILE PHONES, AND MICROCONTROLLERS [8]

Platform	Memory	Storage	Power
Cloud GPU	16 GB HBM	TB/PB	250W
Mobile CPU	4 GB DRAM	64 GB Flash	8W
Microcontroller	2-1024 kB SRAM	32-2048 kB eFlash	0.1-0.3W

TABLE II

MLPerf Tiny v0.5 Inference Benchmarks [9]

Application	Dataset (Input Size)	Model Type (TFLM model size)	Quality Target (Metric)
Keyword Spotting	Speech Commands [10] (49×10)	DS-CNN [11] [12] [13] (52.5 kB)	90% (Top-1)
Visual Wake Words	VWW Dataset [14] (96×96)	MobileNetV1 [12] (325 kB)	80% (Top-1)
Image Recognition	CIFAR-10 [15] (32×32)	ResNetV1 [16] (96 kB)	85% (Top-1)
Anomaly Detection	ToyADMOS [17], MIMII [18] (5×128)	FC-Autoencoder [9] (270 kB)	0.85 AUC

TABLE III

FEATURES OF NOTABLE TINYML DATA ENGINEERING FRAMEWORKS

Framework	Data Type	Collection	Labeling	Alignment	Augmentation	Visualization	Cleaning	Open-source
Edge Impulse [32]	Audio, images, time-series	Real-time (WebUSB, serial daemon, Linux SDK), offline	AI-assisted, DSP-assisted, manual	χ	Geometric image transforms, noise, audio spectrogram transforms, color depth	Images, plots: raw, spectrogram, statistical, DSP, MFE, MFCC, syntiant, feature explorer	Class balancing, crop, scale, split	χ
MSWC [33]	Audio with transcription	Offline speech datasets	Heuristic-based auto	Montreal forced	Synthetic noise, environmental noise	Plots: raw, spectrogram, feature embeddings	Gender balance, speaker diversity, self-supervised quality estimation	✓
SensiML DCL [34]	Time-series, audio	Real-time (WiFi, BLE, Serial daemon), offline	Plot-assisted, Threshold-based auto	Video-assisted	Noise, pool, convolve, drift, dropout, quantize, reverse, time warp	Plots: raw, spectrogram, statistical, DSP, MFCC	Class balancing, crop, scale, split	χ
Qeexo AutoML [35]	Time-series, audio	Real-time (Serial daemon, BLE), offline	Plot-assisted	χ	χ	Plots: raw, spectrogram, statistical, DSP, MFCC, feature embeddings	Segment	χ
Plumerai Data [36]	Images	offline	AI-assisted, manual	χ	Targetted image transforms, oversampling	Images (AI-assisted visual similarity)	Unit tests, failure case identification	χ

IEEE Sens J. Author manuscript; available in PMC 2023 November 15.

TABLE IV

FEATURES OF NOTABLE TINYML MODEL COMPRESSION FRAMEWORKS

Framework	Compression Type	Parameters	Size or Latency Change*	Open-Source
TensorFlow Lite [46]	Post-training quantization	Bit-width (float16, int16, int8), scheme (full-integer, dynamic, float16)	4× smaller, 2-3× speedup [47]	✓
	Quantization-aware training	Bit-width (arbitrary)	Depends on bit-width (upto 8× smaller)	
	Weight pruning	Sparsity distribution (constant, polynomial decay), pruning policy	5-10× smaller [48], 4× speedup [49]	
	Weight clustering	Number of clusters, initial distribution (random, density-based, linear)	3-6× smaller	
QKeras [50]	Quantization-aware training	Bit-width (arbitrary), symmetry, quantized layer definitions, quantized activation functions	Depends on bit-width (upto 8× smaller)	✓
Qualcomm AIMET [51]	Post-training quantization	Bit-width (arbitrary), rounding mode (nearest, stochastic), scheme (data-free, adaptive rounding)	Depends on bit-width (upto 8× smaller)	✓
	Quantization-aware training	Bit-width (arbitrary), scheme (vanilla, range-learning)		
	Channel pruning	Compression ratio, layers to ignore, compression ratio candidates, reconstruction samples, cost metric	2× smaller	
	Matrix factorization	Factorization algorithm (weight SVD, spatial SVD), compression ratio, fine-tuning (per layer, rank rounding)		
Plumerai LARQ [52]	Binarized network training	Bit-width (int1), quantized activation functions, quantized layer definitions (convolution primitives and dense), binarized model backbones	8× smaller, 8.5-19× speedup (with LARQ compute engine) [53]	✓
Microsoft NNI [54]	Post-training quantization	Scheme (naive, observer), bit-width (8-bit, arbitrary), type (dynamic, integer), operator type	Depends on bit-width (upto 8× smaller)	✓
	Quantization-aware training	Scheme (Vanilla, LARQ, learned step size, DoReFa), bit-width (8-bit, arbitrary), type (dynamic, integer), operator type, optimizer		
	Basic pruners	Sparsity distribution, mode (normal, dependency-aware), operator type, training scheme, pruning algorithm (level, L1, L2, FPGM, slim, ADMM, activation APOZ rank, activation mean rank, Taylor FO)	1.4-20× smaller, 1.6-5× speedup	
	Scheduled pruners	All parameters of basic pruners, basic pruning algorithm, scheduled pruning algorithm (linear, AGP, lottery ticket, simulated annealing, auto compress, AMC)	1.1-120× smaller, 1.81-4× speedup	
CMix-NN [55]	Quantization-aware training (mixed precision)	Bit-width (int2, int4, int8), weight quantization type (per-channel, per-layer), batch normalization folding type and delay, memory constraints, quantized convolution primitives	7× smaller	✓
Microsoft SeeDot [56]	Post-training quantization (with autotuned and optimized operators)	Bit-width (8-bit), model (Bonsai [57], ProtoNN [58], Fast-GRNN [59], RNNPool [60]), error metric, scale parameter	2.4-82.2× speedup [56]	✓
Genesis[@] [24]	Tucker decomposition and weight pruning	Rank decomposition, network configuration, sparsity distribution, pruning policy, sensing energy, communication energy	2-109× smaller	✗

* for ~1-4% drop in accuracy over uncompressed models.

@ compression framework for intermittent computing systems.

TABLE V

NEURAL ARCHITECTURE SEARCH FRAMEWORKS TARGETTED TOWARDS MICROCONTROLLERS

Framework	Search Strategy	Hardware Profiling	Inference Engine	Optimization Parameters	Open-Source
SpArSe [86]	Gradient-driven Bayesian	Analytical	uTensor	Error, SRAM, Flash	χ
MCUNet [31] [121]	Evolutionary	Lookup tables, prediction models	TinyEngine (closed-source)	Latency, Error, SRAM, Flash	χ
MicroNets [8]	Gradient-driven	Analytical	Tensorflow Lite Micro	Latency, Error, SRAM, Flash	χ
μNAS [122]	Evolutionary	Analytical	Tensorflow Lite Micro	Latency, Error, SRAM, Flash	✓
THIN-Bayes [123]	Gradient-free Bayesian	Hardware-in-the-loop, analytical	Tensorflow Lite Micro	Latency, Error, SRAM, Flash, Arena size, Energy	✓
iNAS [124]	Reinforcement Learning	Lookup tables, analytical	Accelerated Intermittent Inference (custom)	Latency*, Error, Volatile Buffer, Flash, Power-Cycle Energy$^{@}$	✓

*sum of progres preservation, progress recovery, battery recharge and compute cost

$^{@}$sum of progres preservation, progress recovery, and compute cost

TABLE VI

COMPARISON OF DIFFERENT NAS SEARCH STRATEGIES [5] [131]

Search Strategy	Top-1% Accuracy	Latency^	Model Size (MAC)	Training Cost (GPU hours)	Search Cost (GPU hours)
Reinforcement Learning^v	74%-75.2%	58mS-70 mS	219M-564M	None*, 180N@	40000N-48000N*, None@
Gradient-driven	73.1%-74.9%	71mS	320M-595M	250N-384N	96N-(288+24N)
Evolutionary	72.4%-80.0%	58mS-59mS	230M-595M	1200-(1200+kN)	40
Bayesian^v	73.4%-75.8%	-	225M	None	23N-552N

dataset: ImageNet-1000, backbone network: MBNetV3, k = fine-tuning epoch count

^on Google Pixel1 smartphone, N = Number of deployment scenarios for which different models must be found [5]

^v Techniques based on RL and Bayesian usually have coupled training and search (training cost included with search cost)

* NASNet [128] and MNASNet [132],

@ MBNetV3 Search [133]

TABLE VII

NAS Hardware Profiling Strategies for Microcontrollers

Method	Speed	Accuracy	NAS Frameworks
Real measurements	Slow	High	THIN-Bayes [123], MNASNet [132], One-shot NAS [141]
Lookup tables	Fast-Medium	Medium-High	FBNet [140], Once-for-All [5], MCUNet [31] [121]
Prediction models	Medium	Medium	ProxylessNAS [129], Once-for-All [5], MCUNet [31] [121], LEMONADE [142]
Analytical	Fast	Low	THIN-Bayes [123], MicroNets [8], μNAS [122], SpArSe [86]

TABLE VIII

FEATURES OF NOTABLE TINYML SOFTWARE SUITES FOR MICROCONTROLLERS

Framework	Supported Platforms	Supported Models	Supported Training Libraries	Open-Source	Free
TensorFlow Lite Micro (Google) [167] [46]	ARM Cortex-M, Espressif ESP32, Himax WE-I Plus	NN	TensorFlow	✓	✓
uTensor (ARM) [168]	ARM Cortex-M (Mbed-enabled)	NN	TensorFlow	✓	✓
uTVM (Apache) [130]	ARM Cortex-M	NN	PyTorch, TensorFlow, Keras	✓	✓
EdgeML (Microsoft) [57] [58] [59] [60], [169]–[171]	ARM Cortex-M, AVR RISC	NN, DT, kNN, unary classifier	PyTorch, TensorFlow	✓	✓
CMSIS-NN (ARM) [164]	ARM Cortex-M	NN	PyTorch, TensorFlow, Caffe	✓	✓
EON Compiler (Edge Impulse) [32]	ARM Cortex-M, TI CC1352P, ARM Cortex-A, Espressif ESP32, Himax WE-I Plus, TENSAI SoC	NN, k-means, regressors (supports feature extraction)	TensorFlow, Scikit-Learn	✗	✓
STM32Cube.AI (STMicroelectronics) [172]	ARM Cortex-M (STM32 series)	NN, k-means, SVM, RF, kNN, DT, NB, regressors	PyTorch, Scikit-Learn, TensorFlow, Keras, Caffe, MATLAB, Microsoft Cognitive Toolkit, Lasagne, ConvnetJS	✗	✓
NanoEdge AI Studio (STMicroelectronics) [173]	ARM Cortex-M (STM32 series)	Unsupervised learning	-	✗	✗
EloquentML [72]	ARM Cortex-M, Espressif ESP32, Espressif ESP8266, AVR RISC	NN, DT, SVM, RF, XGBoost, NB, RVM, SEFR (feature extraction through PCA)	TensorFlow, Scikit-Learn	✓	✓
Sklearn Porter [174]	-	NN (MLP), DT, SVM, RF, AdaBoost, NB	Scikit-Learn	✓	✓
EmbML [175]	ARM Cortex-M, AVR RISC	NN (MLP), DT, SVM, regressors	Scikit-Learn, Weka	✓	✓
FANN-on-MCU [176]	ARM Cortex-M, PULP	NN	FANN	✓	✓
SONIC, TAILS[@] [24]	TI MSP430	NN	TensorFlow	✓	✓

[@]inference framework for intermittent computing systems.

IEEE Sens J. Author manuscript; available in PMC 2023 November 15.

TABLE IX

FEATURES OF NOTABLE TINYML ON-DEVICE LEARNING FRAMEWORKS

Framework	Working Principle	Supported Hardware	Tested Application	Network Type	Open-source
Learning in the Wild [183]	**W**: Per-output feature distribution divergence. **H**: Transfer learning on last-layer; sample importance weighing to maximize learning effect. **T**: Gradient norm for sample selection via uncertainty and diversity.	TI MSP430	Image recognition (MNIST, CIFAR-10, GT-SRB)	CNN	✗
TinyOL [184]	**W**: Running mean and variance of streaming input **H**: Transfer learning on additional layer at the output of the frozen network using stochastic gradient descent (SGD).	ARM Cortex-M	Anomaly detection	Autoencoder	✗
ML-MCU [185]	**H**: Optimized SGD (inherits stability of GD and efficiency of SGD); optimized one-versus-one (OVO) binary classifiers for multiclass classification	ARM Cortex-M, Espressif ESP32	Image recognition (MNIST), mHealth (Heart Disease, Breast Cancer), Other (Iris)	Optimized OVO binary classifiers	✓
Train++ [186]	**W**: Confidence score of prediction. **H**: Incremental training via constrained optimization classifier update	ARM Cortex-M, ARM Cortex-A, Espressif ESP32, Xtensa LX	Image recognition (MNIST, Banknote Authentication), mHealth (Heart Disease, Breast Cancer, Haberman's Survival), Other (Iris, Titanic Survival)	Binary classifiers	✓
TinyTL [19]	**H**: Update bias instead of weights and use lite residual learning modules to recoup accuracy loss	ARM Cortex-A	Face recognition (CelebA), Image recognition (Cars, Flowers, Aircraft, CUB, Pets, Food, CIFAR-10, CIFAR-100)	CNN (ProxylessNAS-MB, MBNetV2)	✓
Imbal-OL [187]	**T**: Weighted replay and oversampling for minority classes	ARM Cortex-A	Image recognition (CIFAR-10, CIFAR-100)	CNN (ResNet-18)	✗
QLR-CL [188]	**H**: Continual learning with quantized latent replays (store activation maps at latent replay layer instead of samples), slow-learning below the latent replay layer.	PULP	Image recognition (Core50)	CNN (MBNetV1)	✗

W: When to learn, H: How to learn, T: What to learn (sample selection)

TABLE X

FEATURES OF NOTABLE TINYML FEDERATED LEARNING FRAMEWORKS

Framework	FL Strategy	Communication Stack	Scalability and Heterogeneity	Privacy	Client Hardware (language)	Open-source
Flower [190] [191]	FedAvg, Fault tolerant FedAvg, FedProx, QFedAvg, FedAdagrad, FedYogi, FedAdam	Bidirectional gRPC and *ClientProxy* (language, communication and serialization agnostic)	FedFS (partial work, importance sampling, and dynamic time-outs to handle bandwidth heterogeneity); Virtual Client Engine for scheduling and resource management (15M clients tested)	Salvia secure aggregation	CPU, GPU, MCU (Python, Java, C++)	✓
FedPARL [192]	Reparametrized FedAvg with sample-based pruning	None (simulated)	Resource tracking (memory, battery life, bandwidth, and data volume); Trust value tracking (task completion, delay, model integrity); Partial work (12 clients tested)	Vanilla model aggregation	None (simulated)	χ
DIoT [193]	FedAvg	Bidirectional WebSocket protocol over WiFi and Ethernet	AuDI device-type identification (15 clients tested)	Vanilla model aggregation	CPU, GPU (Python and JavaScript)	χ
PruneFL [194]	FedAvg with adaptive and distributed pruning	WiFi and Ethernet, with distributed pruning to reduce communication overhead	Adaptive pruning to modify local models based on resource availability (9 clients tested)	Vanilla model aggregation	CPU, MCU (Python)	✓
TinyFedTL [195]	FedAvg with last layer transfer learning	USART	9 clients tested	Vanilla model aggregation	MCU (C++)	✓
FLAgr [196]	Reinforcement learning	None (simulated)	Real-time collaboration scheme discovery via deep deterministic policy gradient (1000 clients tested)	Rating feedback mechanism	None (simulated)	χ
PerFit [197]	FedPer, FedHealth, FedAvg, Personalized FedAvg, MOCHA, FedMD, Federated Distillation	WiFi, BLE, Cellular (simulated)	Federated transfer learning, federated distillation, federated meta-learning, and federated multi-task learning to personalize the model, device and statistical heterogeneity (30 clients tested)	Vanilla model aggregation	None (simulated)	χ

TABLE XI

SUMMARY OF IMAGE RECOGNITION FOR MICROCONTROLLERS

Method	Dataset	Accuracy	SRAM (kB)	Flash (kB)	MACs (M)
ResNet8 [9]	CIFAR-10	85%	-	96	25.3
FastRNN [59]	Pixel MNIST-10	96%	<32	166	-
FastGRNN [59]		98%		6	
MCUNetV2-M4 [121]	ImageNet	T1- 65%, T5- 86%	196	1010	119
	Pascal VOC	mAP: 64.6%	247	<1000	172
MCUNetV2-H7 [121]	ImageNet	T1- 72%, T5- 91%	465	2032	256
	Pascal VOC	mAP: 68.3%	438	<2000	343
μNAS CNN [122]	CIFAR-10$^{B, M}$	77-86%	0.9-15.4	0.7-11.4	0.04-0.38
	MNIST	99%	0.49	0.48	0.029
	Fashion MNIST	93%	12.6	63.6	4.4
SpArSe CNN [86]	CIFAR-10$^{B, M}$	73-82%	1.2	0.78	
	MNIST	97-99%	1.3-1.9	1.4-2.8	
	Chars74k^{B}	78%	0.72	0.46	
ProtoNN [58]	CIFAR-10^{B}	76%		15.9	
	WARD^{B}	96%		15.9	
	MNIST$^{B, M}$	96%		16-63.4	
	USPS$^{B, M}$	95-96%		11.6-64	
	CUReT	94%		63.1	
Bonsai [57]	CIFAR-10^{B}	73%		0.5	
	WARD^{B}	96%		0.47	
	MNIST$^{B, M}$	94-97%		0.49-84	
	USPS^{B}	94%		0.5	
	CUReT	95%		115	
	Chars74k^{B,M}	59-75%		0.5-101	
SqueezeNet [92]	ImageNet	T1-58%, T5-80%		470-4800	349-848 [199]
Compressed LeNet [49]	MNIST	98-99%		27-44	-
AttendNets [114]	ImageNet	T1- 72-73%		~ 1000	191-277
AttendSeg$^{\wedge}$ [115]	CamVid	90%		1190	7450
ASL CNN [200]	Kaggle ASL	75-99%	< 400	185	-
Masked Face CNN [201]	Custom Masked Face	99%	< 400	128	
RaScaNet [202]	Pascal VOC^{B}	83-86	4-8	31-46	9.7-56.3
RNNPool MbNetv2 [60]	ImageNet	T1- 70%	240	<2000	226
Batteryless CNN [24]	MNIST	99%	<8	<256	-
FOMO [32]	Beer and Can	96%	244	77.6	

B = binary dataset, M = multiclass dataset (assume M if unspecified).

^semantic segmentation from video.

mAP: mean average precision, T1: top 1%, T5: top 5%.

TABLE XII

SUMMARY OF VISUAL WAKE WORDS DETECTION FOR MICROCONTROLLERS

Method	Accuracy	SRAM (kB)	Flash (kB)	MACs (M)
MobileNetV1 [8]	80%	-	325	15.7
MicroNets MbNetV2 [8]	78-88%	75-275	250-800	-
MCUNetV2 [121]	90-94%	30-118	< 1000	
RaScaNet [202]	88-92%	4-8	15-60	8-57
RNNPool MbNetv2 [60]	86-91%	8-32	250	38-53
MNasNet [14]	85-90%	50-250	400	10-54

Dataset: Visual Wake Words [14].

TABLE XIII

SUMMARY OF AUDIO KEYWORD SPOTTING AND SPEECH RECOGNITION FOR MICROCONTROLLERS

Method	Dataset	Accuracy	SRAM (kB)	Flash (kB)	MACs (M)
DS-CNN [9]	SC*	92%	-	52.5	5.54
MicroNets DS-CNN [8]		96%	103	163	16.7
FastRNN [59]		92%	< 2-32	56	-
	STCI@	97%		8	
FastGRNN [59]	SC	92%		5.5	
	STCI	98%		1	
ShallowRNN [169]	SC	94%	1.5	26.5	0.59
	STCI	99%			0.3
MCUNet DS-CNN [31]	SC	96%	311	<1000	-
μNAS CNN [122]		96%	21	37	1.1
Hello Edge DS-CNN [13]		94%	< 320	38.6	5.4
TinySpeech-Z [113]		92%	-	21.6	2.6
LMU-4 [207]		93%		49	-
Kronecker LSTM [90]		91%		8	0.02
TinyLSTM^ [208]	CHiME2	SDR: 13.0 dB	3.7	310	0.66
ULP RNN^ [209]	SC, MUSAN	<3% NTR	-	0.52	-

*SC refers to the Google Speech Commands dataset.

@STCI refers to the Microsoft STCI Wake Words dataset.

^for speech enhancement.

^for wake-words detection in a noisy environment.

TABLE XIV

SUMMARY OF ANOMALY DETECTORS FOR MICROCONTROLLERS

Method	Dataset	Mean AUC	Peak SRAM (kB)	Flash (kB)	MACs (M)
OutlierNets [120]	MIMII	0.83	-	2.7-26.7	2.87-22.9
MicroNet DS-CNN [8]	ToyADMOS, MIMII	0.96	114-383	253-442	38-129
FC-AE [9]	ToyADMOS, MIMII	0.85	4.7	270	0.52
DROCC [171]*	CIFAR-10	0.74	-	248	1.31
	Thyroid	0.78		1.7	0.00031
	Arrhythmia	0.69		23.2	0.011
	Abalone	0.68		1.9	0.00038
	Epileptic Seizure	0.98		279	-
AnoML CNN [211]	AnoML	0.57	< 256	19.5-19.6	

*DROCC uses an FC-AE for Thyroid, Arrhythmia, and Abalone datasets, LeNet-5 for CIFAR-10, and 1-layer LSTM for Epileptic seizure dataset.

TABLE XV

SUMMARY OF ACTIVITY AND GESTURE DETECTORS FOR MICROCONTROLLERS

Method	Sensor(s)	Task	Accuracy	Flash
GesturePod ProtoNN [212]	MPU6050 inside white cane	Detect 5 cane gestures	92%	6 kB
AURITUS FastRNN [2]	eSense earable	Detect 9 macro activities	98%	6 kB
Bian *et al.* 1D-CNN [213]	Wrist-worn capacitive array	Detect 7 hand gestures	96%	30 kB
T'Jonck *et al.* CNN [214]	BMA400 inside mattress	Detect 5 bed activities	89%	< 1 MB
Zhou *et al.* HDC + SVM [215]	MPU-6050 and EMG pad (wrist-worn)	Detect 13 hand gestures across 8 limb positions	93%	135 kB
Elsts *et al.* CNN [216]	Colibri Wireless IMU	Detect 18 macro activities	73% (F1 score)	20 kB
Coelho *et al.* DT [217]	Chest, waist and ankle-mounted IMU	Detect 12 macro activities	97%	22 kB
FastGRNN (LSQ) [57]	IMU on torso and limbs	Detect 6 macro activities and 19 sports activities	96% 84%	3 kB 3.25 kB

TABLE XVI

EXAMPLE TINYML MHEALTH APPLICATIONS FOR MICROCONTROLLERS

Method	Task	Performance	Flash
TinyEats [221]	Dietary monitoring	Accuracy 95%	35 kB
Arlene *et al.* [222]	Sleep apnea detection	Accuracy: 99%	212 kB
Petrović *et al.* [223]	Cough detection	Accuracy 95%	35 kB
DROCC [171]*	Detect cardiac arrythmia	AUC: 0.69	23.2 kB
	Detect epileptic seizure	AUC: 0.98	279 kB
	Detect hypothyroid condition	AUC: 0.78	1.7 kB
AURITUS BONSAI [2]	Earable fall detection	Accuracy: 98%	2.3 kB

TABLE XVII

SUMMARY OF FACE DETECTORS FOR MICROCONTROLLERS

Method	Dataset	Performance	Peak SRAM	MACs
FaceReID [224]	VGGFace2	Accuracy: 0.97	352 kB	85M
Batteryless Face Detection [25]	CelebA	Accuracy: 0.97	384 kB	-
EagleEye [225]	WIDER FACE	mAP*: 0.77	1170 kB	80M
RNNPool S3FD [60]	WIDER FACE	mAP*: 0.83	225 kB	120M
MCUNetv2 S3FD [121]	WIDER FACE	mAP*: 0.89	672 kB	110M

*mean average precision for ≤3 faces

TABLE XVIII

IMPACT OF FEATURE PROJECTION VS. RAW DATA INFERENCE

Application	Method	Accuracy	Latency/MAC	Flash
Gesture Recognition[V]	CNN [32]	100%	11 mS	45.3 kB
	MLP [32]	100%	5 mS	17.8 kB
Human Activity Recognition[@]	TCN [2]	94.6%	7.52M	52.8kB
	FastGRNN [2]	97.6%	-	13.1 kB
	Bonsai [2]	80.3%	0.0136M	14.8 kB
Anomaly Detection[*]	DROCC [171]	F1 - 68%	0.00038M	1.9kB
	OC-SVM [171]	F1 - 48%	-	2.99 MB

[V]: Device: Cortex M7

[*]Dataset: Abalone

[@]Dataset: AURITUS

Models operating on features

TABLE XIX

IMPACT OF COMPRESSION VS. NO COMPRESSION

Application	Dataset	Method	Accuracy	Latency/MAC	Flash
Human Activity Recognition	HAR-2[V]	FastGRNN [59]	94.5%	-	29 kB
		FastGRNN-L [59]	96.8%	-	28 kB
		FastGRNN-LS [59]	96.3%	172 mS	17 kB
		FastGRNN-LSQ [59]	95.6%	62 mS	3 kB
	HAR-1[@]	BiLSTM [228]	91.9%	470 mS	1.5 MB
		BiLSTM-Prune [90]	83%	98.2 mS	76 kB
		BiLSTM-Q [90]	91.1%	-	384 kB
		BiLSTM-KP [90]	91.1%	157 mS	75 kB
Audio Keyword Spotting	Speech Commands[*]	FastGRNN [59]	93.2%	-	57 kB
		FastGRNN-L [59]	93.8%	-	41 kB
		FastGRNN-LS [59]	92.6%	779 mS	22 kB
		FastGRNN-LSQ [59]	92.2%	242 mS	5.5kB
		LSTM [13]	92.5%	26.8 mS	243 kB
		LSTM-Prune [90]	84.9%	5.9mS	16 kB
		LSTM-Q [90]	92.2%	-	65 kB
		LSTM-KP [90]	91.2%	17.5 mS	15 kB
Image Recognition	MNIST-10[@]	Bonsai [57]	97%	-	84 kB
		Bonsai-Q [57]	~97%	-	21 kB
		LSTM [90]	99.4%	6.3 mS	45 kB
		LSTM-Prune [90]	96.5%	0.7 mS	4.2 kB
		LSTM-KP [90]	98.4%	4.6 mS	4.1 kB
	ImageNet	SqueezeNet [92]	T1-57.5%	848M	4.8 MB
		SqueezeNet-PQE [92]	T1-57.5%	349M	0.5 MB
		AlexNet [57]	T1-57.2%	723M	240 MB
		AlexNet-Prune [57]	57.2%	-	27 MB
		AlexNet-PQ [57]	57.2%	-	9 MB
		AlexNet-PQE [57]	57.2%	~348M	6.9 MB

[V] Device: SAM3X8E Cortex-M3,

[@] Hikey 960 Cortex A73

[*] Device: SAM3X8E Cortex-M3 for the first four, Hikey 960 Cortex A73 for the last four

Q - Quantized, S - Sparsified, L - Low Rank Factorization,

KP - Kronecker Products, E - Huffman Encoding

Uncompressed and lightweight model,

Uncompressed and vanilla model,

Compressed and lightweight model,

Compressed and vanilla model

TABLE XX

IMPACT OF LIGHTWEIGHT VS. VANILLA MODEL USAGE

Application	Dataset	Method	Accuracy	Latency/MAC	Flash
Human Activity Recognition	HAR-2 [v]	FastRNN [59]	94.5%	<172 mS	29 kB
		FastGRNN [59]	95.4%	172 mS	29 kB
		RNN [59]	91.3%	590 mS	29 kB
		LSTM [59]	93.7%	OOM [^]	74 kB
	AURITUS [*]	TCN [2]	95.12%	1380 mS	60 KB
		SVM [2]	99.9%	OOM	23 MB
		MLP [2]	99.8%	OOM	418 kB
		Coarse DT [2]	98.5%	OOM	1100 kB
		AdaBoost [2]	98.7%	OOM	81.6 MB
Audio Keyword Spotting	Speech Commands	DS-CNN [9]	92%	5.54M	52.5 kB
		TinySpeech-Z [113]	92.4%	2.6M	21.6 kB
		LMU-4 [207]	92.7%	-	49 kB
		CNN [229]	90.7%	76M	556 kB
Image Recognition	MNIST-10	Bonsai [57]	97%	-	84 kB
		ProtoNN [58]	95.9%		63 kB
		kNN [57]	94.3%	OOM	184 MB
		MLP [57]	98.3%	OOM	3.1 MB
	ImageNet	AttendNets [114]	T1-71.7%	191M	~1 MB
		SqueezeNet [@] [92]	T1-57.5%	848M	4.8 MB
		AlexNet [93]	T1-57.2%	723M	240 MB

[v] Device: SAM3X8E Cortex-M3,

[*] Device: STM32 Cortex-M4 and M7

[^] OOM = Out of memory on tested microcontrollers

[@] Uncompressed

■ Vanilla models

TABLE XXI

IMPACT OF NAS VS. HANDCRAFTED MODELS

Application	Dataset	Method	Accuracy	Latency/MAC	Flash
Inertial Odometry[o]	OxIOD	L-IONet TCN [231]	2.82 m	13.9M	183 kB
		RoNiN TCN [232]	0.42m	220M	2.1 MB
		TinyOdom TCN [59]	1.24m-1.37 m	4.64M-8.92M	71 kB-118 kB
Audio Keyword Spotting	Speech Commands	DS-CNN [9]	92%	5.54M	52.5 kB
		MicroNets DS-CNN [8]	95.3% - 96.5%	16M-129M	102 kB-612 kB
		μNAS-CNN [122]	95.4-95.6%	1.1M	19 kB-37 kB
Image Recognition	Visual Wake Words[*]	MBNetv2 [8]	86%	0.46s	375 kB
		MicroNets MBNetv2 [8]	78.1%-88%	0.08s-1.13s	230 kB-833 kB
	MNIST-10[*]	Bonsai [57]	94.4%	8.9 mS	1.97 kB
		SpArSe-CNN [86]	95.8%-97%	27mS-286mS	2.4kB-15.9kB
	CIFAR-10[B,*]	Bonsai [57]	73%	8.2 mS	1.98 kB
		SpArSe-CNN [86]	70.5%-73.4%	0.49s-2.52s	2.7 kB-9.9 kB
		μNAS-CNN [122]	77.5%	-	0.69kB

[o] Accuracy metric is relative trajectory error [232] (lower is better)

[*] Device: STM32 Cortex-M4 and M7

[B] Binary dataset

Handcrafted models

TABLE XXII

IMPACT OF RUNTIME OPTIMIZATIONS VS. NO OPTIMIZATIONS

Application	Dataset	Method	Accuracy	Latency/MAC	Flash
Human Activity Recognition	Custom*	CNN-TFLM [189]	85%	58 mS	275 kB
		CNN-Cube.AI [189]	85%	14 mS	192 kB
Audio Keyword Spotting	Speech Commands*	CNN-TFLM [189]	-	380 mS	288 kB
		CNN-Cube.AI [189]		373 mS	247 kB
Image Recognition	ImageNet	MCUNet MbNetv2 [121]	60.3%-68.5%	68M-126M	1MB-2MB
		MCUNetV2 MbNetv2 [121]	64.9%-71.8%	119M-256M	1MB-2MB
	Pascal VOC	MbNetv2+CMSIS [121]	mAP: 31.6%	34M	OOS
		MCUNetV MbNetv2 [121]	mAP: 51.4%	168M	<2 MB
		MCUNetV2 MbNetv2 [121]	mAP: 64.6%	172M	<1 MB
	CIFAR-10@	CNN [164]	80.3%	456 mS	< 1 MB
		CNN-CMSIS [164]	80.3%	99 mS	< 1 MB

*Device: STM32 Cortex-M4

@Device: STM32 Cortex-M7, OOS: Overflowed SRAM

■ Superior optimization techniques than comparing method in the same dataset class

TABLE XXIII

IMPACT OF ONLINE LEARNING VS. STATIC MODELS

Application	Dataset	Method	Accuracy	Latency
Image Recognition *	MNIST-10	CNN [183]	10%-82%	-
		CNN (LW) [183]	65%-98%	
	CIFAR-10	CNN [183]	12%-38%	
		CNN (LW) [183]	55%-68%	
Anomaly Detection @	Custom	Autoencoder [184]	75%	1.75 mS
		Autoencoder (TinyOL) [184]	100%	1.92 mS

* Device: TI MSP430

@ Signal reconstruction error, Device: nRF52840 Cortex-M

No online learning