Fitting Machine Learning into Kilobytes: An Empirical Study of Quantization, Pruning and Model Reduction for Microcontroller-Class IoT Devices

Shraddha Gupta[1], Shilpa Jaiswal[2]
[1,2]*Shri Vaishnav Institute of Computer Applications, Shri Vaishnav Vidyapeeth Vishwavidyalaya, Indore, Madhya Pradesh, India*

***Abstract*—Running machine learning directly on low-cost microcontrollers saves energy and protects privacy in Internet of Things devices, but these chips offer only kilobytes of memory. This paper measures how quantization, pruning and structural reduction affect the accuracy, compiled size and robustness of seven classifier families: logistic regression, a multilayer perceptron, k-nearest neighbours, a decision tree, a random forest, gradient boosting and a support vector machine. Twenty-four configurations were trained on three public datasets, exported to standalone C, compiled for an Arm Cortex-M4F and an 8-bit AVR ATmega328P, and verified to give exactly the same predictions as their Python references. Accuracy was estimated over 15 cross-validation folds and robustness under Gaussian noise and pixel dropout. Storing parameters in 8 bits never caused a significant loss of accuracy, reduced flash by up to 3.9 times and, because the input quantizer clips out-of-range values, often improved robustness. Four-bit weights were harmless on the two small tabular tasks but cost the neural network 14.7 percentage points on handwritten digits, and pruning 90% of the weights reduced accuracy on every dataset. Replacing the nearest-neighbour training set with ten prototypes per class reached 98.3% accuracy on digits in 6.5 KB of flash, within the budget of an Arduino-class board. The models most accurate on clean data, the support vector machine and the neural network, were the most fragile under noise. Practical guidance is given for 8-bit and 32-bit devices.**

***Index Terms*—TinyML, model compression, quantization, pruning, microcontrollers, edge computing**

I. INTRODUCTION

The Internet of Things (IoT) places sensors in farms, factories, hospitals and homes. Most of these sensors are attached to small microcontrollers that run on batteries for months and cost very little. Sending every raw reading to the cloud for analysis costs energy, needs a reliable network link and raises privacy concerns, so there is growing interest in running machine learning (ML) inference directly on the device, at the edge of the network [1]. This practice, often called tiny machine learning (TinyML), is now an active research area [2]– [5].

The obstacle is size. A typical 8-bit microcontroller such as the ATmega328P, used on the Arduino Uno, has 32 KB of flash memory for the program and 2 KB of SRAM for working data, and no hardware floating-point unit [6]. Even a 32-bit device such as the nRF52840, with an Arm Cortex-M4F core, has only 1 MB of flash and 256 KB of RAM [7]. A random forest or a support vector machine trained with default settings can easily exceed these limits. Model compression, which covers quantization of numbers to fewer bits, pruning of unimportant weights and reduction of the model structure, is the standard answer [8]– [10].

Most compression research, however, targets deep neural networks for images and speech, and reports model size as an estimate of parameter count or bytes [11], [12]. Many practical IoT tasks are small tabular or low-resolution problems where classical models such as logistic regression, decision trees, nearest neighbours and small multilayer perceptron's remain competitive. For these models, practitioners still lack a simple, measured answer to three questions: how much accuracy each compression method costs, how large the resulting program really is on a specific microcontroller, and whether compression makes the model more fragile when sensor readings are noisy [13].

This paper answers these questions with an end-to-end empirical study. We train seven model families on three public datasets, apply 17 compression variants, generate standalone C code for every configuration and compile it for two real microcontroller targets, the Cortex-M4F and the 8-bit AVR ATmega328P.

The study is guided by four research questions:
RQ1 (Accuracy). How much accuracy is lost by 8-bit and 4-bit quantization, weight pruning and structural reduction of each model family?
RQ2 (Footprint). How much flash and RAM does each compiled model actually need, and which models fit an 8-bit board with 32 KB of flash and 2 KB of SRAM?
RQ3 (Robustness). Does compression make a model more sensitive to noisy or missing sensor inputs?
RQ4 (Trade-off). Which combinations of model family and compression give the best accuracy for a given memory budget?

The contributions of this paper are as follows:
1. A reproducible pipeline that trains, compresses, exports to C, cross-compiles and verifies 24 model configurations, with every compiled model checked to give exactly the same predictions as its Python reference.
2. Measured flash, RAM and stack figures for two microcontroller targets, rather than parameter counts alone.
3. An evaluation of accuracy over 15 cross-validation folds and of robustness under three types of input corruption.
4. Practical guidance on which compression to choose for 8-bit and 32-bit IoT devices.

The rest of the paper is organised as follows. Section II reviews related work. Section III describes the datasets, models, compression methods and measurement pipeline. Section IV presents the results, Section V discusses their implications and limitations, and Section VI concludes.

II. RELATED WORK

A. TinyML and edge inference

Edge computing moves computation close to where data is produced, to reduce latency, bandwidth and privacy risk [1]. Warden and Situnayake [2] popularised the term TinyML for ML on microcontrollers, and recent reviews describe its tools, applications and open problems [3]– [5]. TensorFlow Lite Micro provides an interpreter for running quantized neural networks on such devices [14], and the MLPerf Tiny benchmark defines standard tasks and measurement rules for them [12]. MCUNet showed that jointly designing the network and the inference engine can fit image classification into the memory of a microcontroller [11].

B. Classical models for resource-scarce devices

Classical models are often a better fit for the smallest devices. Bonsai learns a single shallow, sparse tree that runs in 2 KB of RAM [15], and ProtoNN compresses *k*-nearest-neighbour classification into a small set of learned prototypes [16]. Older ideas point the same way: the condensed nearest-neighbour rule removes redundant training points [17], and *k*-means [18] can summarise each class by a few centroids. Decision trees [19], random forests [20] and gradient-boosted trees [21], [22] need only comparisons at inference time, which suits processors without floating-point hardware.

C. Quantization and pruning

Quantization represents weights and activations with low-precision integers. Jacob *et al.* [23] introduced integer-arithmetic-only inference with 8-bit weights and activations and 32-bit accumulators, the scheme used by most microcontroller runtimes; practical guides and surveys are given in [9], [10], [24]. Pruning removes weights of small magnitude and fine-tunes the remaining ones [25], and combined with quantization it can shrink networks dramatically [8]. Blalock *et al.* [26] warn that pruning results are hard to compare because studies rarely report the same metrics, and the lottery ticket hypothesis [27] suggests that sparse sub-networks can train well when found early. Energy studies show that moving data from memory costs far more than arithmetic [28], [29], so smaller models also tend to be more energy-efficient [30].

D. Research gap

Hooker *et al.* [13] found that compression can change the predictions on a small subset of inputs even when overall accuracy is unchanged, and robustness to corrupted inputs is a known weakness of ML models [31]. Yet most compression studies report accuracy

and parameter bytes only, for a single model family. Few studies compare classical and neural models under the same compression budget, measure the compiled size on real microcontroller instruction sets, verify that the deployed code matches the trained model, and test robustness to noisy inputs together. This paper fills that gap for small tabular and image tasks typical of IoT sensing.

III. MATERIALS AND METHODS

A. Target microcontrollers

We use two reference targets that represent the two ends of the microcontroller range (Table I). The ATmega328P is an 8-bit AVR device with 32 KB of flash and 2 KB of SRAM and no floating-point unit, so all floating-point arithmetic must be emulated in software [6]. The nRF52840 is a 32-bit Arm Cortex-M4F device with a single-precision floating-point unit, 1 MB of flash and 256 KB of RAM [7].

Table I Reference microcontroller targets

Property	ATmega328P	nRF52840
Core	8-bit AVR	32-bit Arm Cortex-M4F
Maximum clock	20 MHz	64 MHz
Flash / SRAM	32 KB / 2 KB	1 MB / 256 KB
Floating point	software only	single-precision FPU
Typical board	Arduino Uno	Arduino Nano 33 BLE

B. Datasets

Three public datasets that ship with scikit-learn [32] are used, so all results can be reproduced without downloading data (Table II). *DIGITS* contains 1,797 handwritten digits scanned as 8 × 8 images with pixel intensities from 0 to 16 [33], a stand-in for a low-resolution image or tactile sensor. *WINE* contains 13 chemical measurements of 178 wines from three cultivars [34], typical of a chemical or gas sensor array.

WDBC contains 30 features computed from digitised images of 569 breast-mass aspirates, labelled benign or malignant [35], typical of a diagnostic device.

Table II Datasets

Dataset	*n*	*d*	Classes	Sensor analogue
DIGITS [33]	1,797	64	10	8 × 8 image sensor
WINE [34]	178	13	3	chemical sensor array
WDBC [35]	569	30	2	diagnostic device

C. Input quantization

Sensors deliver integers from an analogue-to-digital converter, so all integer configurations take 8-bit inputs. For each feature j, the minimum l_j and maximum u_j are taken from the training data and each value is mapped to an unsigned 8-bit code:

$$q_j = \mathrm{clip}\left(\mathrm{round}\left(\frac{x_j - l_j}{s_j}\right), 0, 255\right), \quad s_j = \frac{u_j - l_j}{255} \tag{1}$$

Any standardisation that a model needs is folded into its weights, so the device never has to compute means or standard deviations.

D. Model families

Seven families are compared, all trained with scikit-learn [32] using fixed hyperparameters chosen before the experiments and not tuned on test data: multinomial logistic regression (LR) [36] with L2 penalty $C = 1$; a multilayer perceptron (MLP) with one hidden layer of 32 ReLU units, trained with Adam [37] and weight decay 10^{-3}; k-nearest neighbours (kNN) [38] with $k = 5$ on min–max scaled features; a decision tree (DT) [19] of depth at most 8; a random forest (RF) [20] of 100 unpruned trees with majority voting; histogram-based gradient boosting (GB) [21], [22] with 100 iterations of 15-leaf trees and learning rate 0.1; and a support vector machine (SVM) [39] with a radial basis function kernel, $C = 10$ and one-vs-one voting as in LIBSVM [40]. The full-precision (fp32) version of each family is its baseline.

E. Compression methods

Weight quantization. For LR and MLP, the real-valued weights w of each output unit c are quantized symmetrically to b bits with their own scale:

$$\hat{w}_{cj} = \mathrm{round}\left(\frac{w_{cj}}{\Delta_c}\right), \quad \Delta_c = \frac{\max_j |w_{cj}|}{2^{b-1} - 1} \tag{2}$$

with $b = 8$ (int8) or $b = 4$ (int4, two weights packed per

byte). Products of integer weights and 8-bit inputs are summed in a 32-bit integer accumulator a_C, and only the final score $z_C = \Delta_C a_C + \beta_C$ is computed in floating point. In the int8 MLP, the hidden activations are also stored as 8-bit codes, following integer-arithmetic-only inference [23]:

$h_k = \text{clip}(\lfloor M_k (a_k + B_k) + 0.5 \rfloor, 0, 255)$ (3)

where B_k is an integer bias, $M_k = \Delta_k / s_h$ is a rescaling factor and the activation scale s_h is calibrated as the 99.9th percentile of the hidden activations on the training data divided by 255.

The clip at zero also implements the ReLU.

Pruning. For the MLP, the weights of each layer with the smallest magnitudes are set to zero at sparsity $s \in \{50\%, 75\%, 90\%\}$, and the network is fine-tuned for 150 epochs with the zeroed weights held at zero [25]. The pruned network is then quantized to int8 and stored in compressed sparse row form, with one 8-bit value and one 8-bit column index per remaining weight.

Prototype reduction. kNN normally stores the whole training
set. We first store it as 8-bit codes (kNN-u8), and then replace it by $m \in \{10, 3\}$ k-means centroids per class [16], [18] used with a 1-nearest-neighbour rule (Proto10, Proto3).

Tree quantization and reduction. Each tree node stores a feature index (1 byte), a threshold and two 16-bit child indices. In the u8 variants, trees are trained directly on the 8-bit codes of (1), so every threshold is an integer between 0 and 255 and is stored in one byte instead of four, and comparisons need no floating point. RF is further reduced to 20 trees of depth 8 (RF20d8) and 10 trees of depth 6 (RF10d6), and GB to 25 iterations (GB25).

SVM. Support vectors are stored as 8-bit codes, with the feature scaling and kernel width folded into one floating-point weight per feature.

Table III lists all 24 configurations.

Table III Model configurations

Family	Baseline	Compressed variants
LR	fp32	int8, int4 weights
MLP	fp32	int8, int4; pruned 50/75/90% + int8
kNN	fp32, $k = 5$	u8; 10 or 3 prototypes per class (u8)
DT	fp32, depth 8	u8 thresholds
RF	fp32, 100 trees	u8; 20 trees depth 8; 10 trees depth 6
GB	fp32, 100 iter.	u8; u8 with 25 iterations
SVM	fp32, RBF	u8 support vectors

F. Code generation, compilation and verification

Each trained configuration is written out as a self-contained C file holding its parameters as constant arrays and one predict () function, with no dynamic memory and no external library except expf for the SVM kernel. Fig. 1 summarises the pipeline. The C file is compiled with Clang 18 from the LLVM project [41] for two targets: Cortex-M4F (Thumb-2, hardware floating point) and AVR ATmega328P, where the parameter arrays are placed in flash with the flash qualifier. Both use size optimisation (-Os). Flash is measured as the sum of the code, read-only data and initialised data sections reported by llvm-size, and the stack frame of predict () is reported by the compiler (-fstack-usage). Run-time library routines, such as software floating-point arithmetic on the AVR and expf, are shared by any program and are not included.

To check that the generated code is correct, the same C file is compiled for the host with GCC and a test harness that classifies every test sample. A configuration is accepted only if its C predictions are identical to the predictions of its Python reference implementation, which reproduces the integer and single-precision arithmetic of the C code. Floating-point contraction is disabled on all targets (-ffp-contract=off) so that results are bit-exact. The harness also gives two host-side cost indicators: the number of x86-64 instructions executed inside predict (), counted with Valgrind/Callgrind [42], and the median wall-clock time per inference on an Intel Xeon core at 2.8 GHz. These are proxies for relative cost only, since execution time on a microcontroller depends on its instruction set and memory.

G. Evaluation protocol

Accuracy is estimated with stratified 5-fold cross-validation repeated three times (15 train–test splits). Input quantizers, scalers, prototypes and calibration statistics are fitted on the training part of each split

only. We report accuracy, macro-averaged F1 score and the *agreement* of each compressed model with its fp32 baseline, the share of test samples on which both predict the same class; one minus the agreement is the share of predictions that compression changed [13].

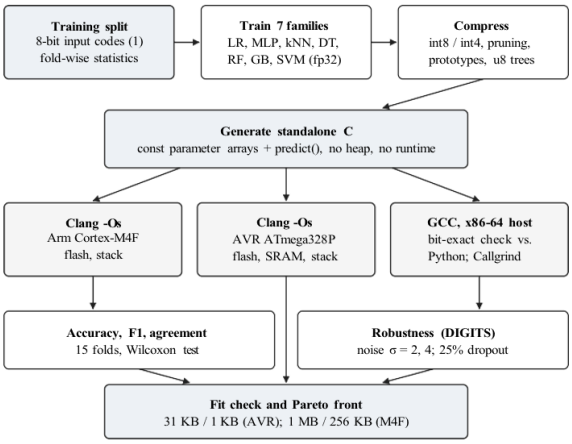

Fig. 1. Measurement pipeline. Every configuration is compiled for two microcontrollers and accepted only if its C predictions match the Python reference exactly.

Each compressed variant is compared with its baseline using a paired Wilcoxon signed-rank test over the 15 folds [43], [44]. Memory footprint is measured on one stratified 80/20 split (seed 42).

Robustness (RQ3) is tested on DIGITS, where corruptions have a clear physical meaning. Each test image is corrupted in three ways: additive Gaussian noise with standard deviation 2 or 4 intensity levels (on the 0–16 scale, rounded and clipped), and random dropout of 25% of the pixels, set to zero, as from dead sensor elements [31]. The models are not retrained on corrupted data.

A configuration is said to *fit* a target if its measured flash is within the device flash and its RAM need, the input buffer plus the static data and stack frame of predict (), is within the device SRAM. For the ATmega328P we reserve 1 KB of flash for the bootloader and run-time libraries, leaving a budget of 31 KB, and 1 KB of SRAM for the rest of the application, leaving 1 KB.

IV. RESULTS

All 72 compiled models (24 configurations × 3 datasets) passed the verification step: their C predictions matched the Python reference on every test sample. The accuracies below are therefore the accuracies of the code that would run on the device. The full pipeline ran on a two-core virtual machine in about 40 minutes.

A. Accuracy of compressed models (RQ1)

Table IV reports accuracy for all configurations. The main result is that 8-bit storage was free. For every family on every

TABLE IV Accuracy (%) over 15 cross-validation fold

Configuration	DIGITS	WINE	WDBC
LR-fp32	97.0 ± 0.8	98.9 ± 1.4	97.6 ± 1.3
LR-int8	97.0 ± 0.8	98.7 ± 1.8	97.9 ± 1.4
LR-int4	96.3 ± 1.1 ↓	98.1 ± 2.0	97.5 ± 1.6
MLP-fp32	97.4 ± 0.7	97.6 ± 2.8	97.4 ± 1.4
MLP-int8	97.3 ± 0.8	97.6 ± 2.6	97.4 ± 1.4
MLP-int4	82.7 ± 6.1 ↓	97.9 ± 2.0	97.1 ± 1.6
MLP-P50-int8	97.3 ± 0.8	97.4 ± 2.5	97.6 ± 1.1
MLP-P75-int8	96.2 ± 0.8 ↓	97.2 ± 2.4	97.4 ± 1.2
MLP-P90-int8	82.1 ± 5.2 ↓	85.8 ± 10.6 ↓	94.9 ± 1.8 ↓
kNN-fp32	98.6 ± 0.7	95.9 ± 3.5	96.8 ± 1.5
kNN-u8	98.6 ± 0.7	95.9 ± 3.5	96.9 ± 1.5
Proto10-u8	98.3 ± 0.6 ↓	96.2 ± 3.3	96.0 ± 1.9
Proto3-u8	96.4 ± 0.8 ↓	97.0 ± 3.1	95.8 ± 1.7
DT-fp32	84.2 ± 2.2	90.1 ± 5.2	92.9 ± 2.3
DT-u8	84.2 ± 2.2	90.3 ± 5.0	93.0 ± 2.0
RF100-fp32	97.6 ± 0.6	98.3 ± 2.1	96.0 ± 2.0
RF100-u8	97.7 ± 0.6	98.5 ± 2.1	96.0 ± 2.1
RF20d8-u8	95.2 ± 1.2 ↓	97.2 ± 3.0	95.4 ± 2.2
RF10d6-u8	90.4 ± 1.5 ↓	97.2 ± 3.2	95.1 ± 2.1 ↓
GB100-fp32	97.4 ± 0.5	97.6 ± 3.0	96.8 ± 1.2
GB100-u8	97.4 ± 0.5	97.0 ± 3.1	96.4 ± 1.6
GB25-u8	95.9 ± 0.7 ↓	96.6 ± 3.7	95.5 ± 2.0 ↓
SVM-fp32	98.3 ± 0.6	98.3 ± 2.1	97.3 ± 1.2
SVM-u8	98.3 ± 0.6	98.3 ± 2.1	97.3 ± 1.2

Mean ± SD. Baselines in bold. ↓/↑: significantly lower/higher than the family baseline (paired Wilcoxon test, $p < 0.05$).

Going below 8 bits depended strongly on the task. LR-int4 lost 0.71 pp on DIGITS ($p = 0.005$) and nothing significant elsewhere, but MLP-int4 lost 14.7 pp on DIGITS while remaining unchanged on WINE and WDBC. Fig. 2(a) shows why. On WINE and WDBC, three weight bits were enough to stay within 1 pp of

the int8 model, whereas DIGITS, with 64 inputs and 10 classes, needed six bits to stay within 0.5 pp; at four bits it fell to 82.7% and at three bits to 26.5%.

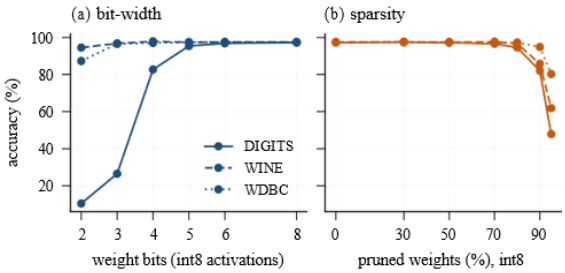

Fig. 2. MLP accuracy (mean over 15 folds) as (a) the weight bit-width is reduced after training, with 8-bit activations, and (b) more weights are pruned before int8 quantization. The point at 0% sparsity is the dense int8 model.

Pruning behaved similarly (Fig. 2(b)). Removing up to half of the MLP weights caused no significant loss on any dataset, and the loss stayed below 1 pp up to 70% sparsity on all three. Beyond 80% accuracy fell quickly, and at 90% sparsity the losses were 15.4 pp (DIGITS), 11.7 pp (WINE) and 2.5 pp (WDBC), all significant.

Because a pruned weight must be stored with its index, pruning at 50% gave no saving in flash over the dense int8 model (3,158 versus 2,976 bytes on DIGITS); savings appeared only at higher sparsity.

Structural reduction was the most effective way to shrink the large models, at a small and predictable cost. Replacing the kNN training set by 10 prototypes per class lost only 0.35 pp on DIGITS (98.3% versus 98.6%, $p = 0.038$) and nothing significant on WINE and WDBC. Shrinking the random forest to 20 trees of depth 8 cost 2.4 pp on DIGITS, and to 10 trees of depth 6 cost 7.2 pp; reducing gradient boosting from 100 to 25 iterations cost 1.5 pp on DIGITS and 1.4 pp on WDBC.

B. Measured footprint (RQ2)

Table V gives the compiled sizes. The generated code itself was small: the predict () function occupied 112–480 bytes on the Cortex-M4F and 168–1,008 bytes on the 8-bit AVR, so for every model larger than 2 KB at least 87% of the flash was taken by parameters. Model size is therefore almost entirely a question of how many parameters are stored and in what format. Stack frames were between 16 and 152 bytes on the Cortex-M4F.

Table V Measured flash and RAM of the compiled models

Configuration	M4 DIG	M4 WIN	M4 WDB	AVR DIG	RAM DIG	Fit AVR
LR-fp32	2.7	0.3	0.4	2.9	278	✓✓✓
LR-int8	0.8	0.2	0.2	1.1	88	✓✓✓
LR-int4	0.6	0.2	0.2	0.8	88	✓✓✓
MLP-fp32	9.6	2.4	4.4	10.0	408	✓✓✓
MLP-int8	2.9	1.0	1.5	3.3	120	✓✓✓
MLP-int4	1.8	0.8	1.1	2.2	120	✓✓✓
MLP-P50-int8	3.1	1.2	1.7	3.4	122	✓✓✓
MLP-P75-int8	1.9	1.0	1.2	2.3	122	✓✓✓
MLP-P90-int8	1.2	0.8	0.9	1.6	122	✓✓✓
kNN-fp32	361	7.8	54	n/c	n/c	✗✓✗
kNN-u8	92	2.2	14	n/c	n/c	✗✓✓
Proto10-u8	6.5	0.6	0.8	6.9	113	✓✓✓
Proto3-u8	2.1	0.3	0.3	2.4	113	✓✓✓
DT-fp32	1.6	0.3	0.5	1.7	288	✓✓✓
DT-u8	1.1	0.2	0.3	1.2	88	✓✓✓
RF100-fp32	299	17	32	n/c	n/c	✗✓✗
RF100-u8	200	12	22	n/c	n/c	✗✓✓
RF20d8-u8	26	2.3	4.3	26	91	✓✓✓
RF10d6-u8	6.5	1.2	2.0	6.7	91	✓✓✓
GB100-fp32	138	16	18	138	314	✗✓✓
GB100-u8	107	13	14	108	122	✗✓✓
GB25-u8	36	4.1	3.6	36	122	✗✓✓
SVM-fp32	205	3.9	11	n/c	n/c	✗✓✓
SVM-u8	71	1.7	3.2	71	2968	✗✓✓

Flash in KB for Cortex-M4F (M4) and ATmega328P (AVR); AVR RAM in bytes (input buffer + static data + stack of predict ()). Fit: fits the ATmega328P budget (31 KB flash, 1 KB RAM) for DIGITS / WINE / WDBC. n/c: cannot be compiled for AVR (an array exceeds its 16-bit address space). Every configuration fits the nRF52840.

Quantization reduced flash by a factor that depends on how much of the model is made of numbers rather than structure. On DIGITS, 8-bit storage shrank LR by 3.1 times (2,724 to 868 bytes), the MLP by 3.3 times, kNN by 3.9 times and the SVM by 2.9 times, but trees only by 1.3–1.5 times, because each node still stores its feature and child indices. Every configuration fitted the 1 MB flash and 256 KB RAM

of the nRF52840. The 8-bit ATmega328P was much more selective. All 24 configurations fitted for WINE and 22 for WDBC, where quantization moved the 100-tree forest from 32.0 KB, just above the 31 KB budget, to 22.0 KB. For DIGITS only 15 fitted: the full kNN, both 100-tree forests, all gradient-boosted models and both SVMs were too large, and the AVR compiler could not even represent the parameter arrays of the largest ones in its 16-bit address space. The DIGITS SVM-u8 would also need about 3 KB of RAM for its kernel values, more than the device has.

The u8 tree models had a further practical advantage on the AVR: DT-u8 and the u8 random forests compiled to pure integer code that called no run-time library at all, while every model that finishes its computation in floating point, including the int8 LR and MLP, needed the software floating-point routines of the AVR. On the host, the instruction counts spanned nearly four orders of magnitude on DIGITS, from 137 instructions per inference for the decision tree and 1,474 for RF20d8 to 59,000 for Proto10-u8 and about 846,000 for the full kNN-u8, which must scan all 1,437 stored images.

C. Robustness to corrupted inputs (RQ3)

Table VI and Fig. 3 show that robustness depended far more on the model family than on compression. Under Gaussian noise with σ = 2, tree ensembles and nearest-neighbour models kept more than 93% accuracy, whereas the fp32 LR fell to 71.7%, the MLP to 28.0% and the SVM to 12.6%. Random dropout of 25% of the pixels was less harmful for these three models and more harmful for the single decision tree, which fell to 43.9%.

Table VI Robustness on DIGITS: accuracy (%) under input corruption

Configuration	Clean	σ = 2	σ = 4	Drop
LR-fp32	97.0	71.7	42.8	76.8
LR-int8	97.0	86.0	70.1	76.9
LR-int4	96.3	85.1	69.4	75.4
MLP-fp32	97.4	28.0	17.5	79.6
MLP-int8	97.3	32.9	23.8	79.8
MLP-int4	82.7	29.6	22.4	63.0
MLP-P75-int8	96.2	49.6	37.8	72.4
MLP-P90-int8	82.1	48.5	37.7	51.9
kNN-fp32	98.6	97.8	87.8	87.8
kNN-u8	98.6	98.3	96.6	87.9
Proto10-u8	98.3	97.8	95.2	87.5
Proto3-u8	96.4	95.5	92.3	86.3
DT-fp32	84.2	61.0	46.5	43.9
DT-u8	84.2	61.1	46.4	43.9
RF100-fp32	97.6	95.1	87.3	86.0
RF100-u8	97.7	95.0	87.3	86.0
RF20d8-u8	95.2	89.1	78.3	78.4
RF10d6-u8	90.4	80.5	68.0	68.8
GB100-fp32	97.4	93.4	81.4	78.9
GB100-u8	97.4	93.4	81.4	78.9
GB25-u8	95.9	88.4	73.4	73.4
SVM-fp32	98.3	12.6	10.4	84.2
SVM-u8	98.3	12.6	10.5	84.3

Mean over 15 folds. σ: Gaussian noise in intensity levels (0–16 scale); Drop: 25% of pixels set to zero. Baselines in bold.

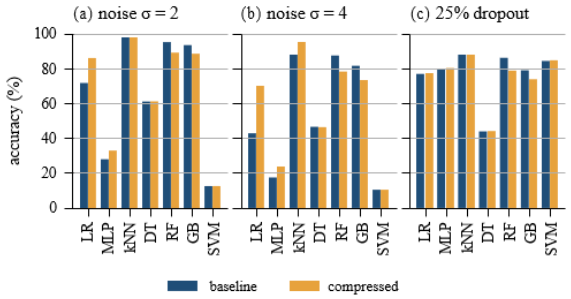

Fig. 3. Accuracy on corrupted DIGITS inputs for each fp32 baseline and one compressed variant: LR-int8, MLP-int8, Proto10-u8, DT-u8, RF20d8-u8, GB25-u8 and SVM-u8.

The collapse of the SVM and MLP has a simple cause. Several border pixels of DIGITS are almost always blank: 12 of the 64 pixels have a training standard deviation below 0.5, and the smallest non-zero one is 0.024. Standardisation divides by this value, so a noise of 2 intensity levels in such a pixel becomes a shift of more than 80 standard deviations, which drives every RBF kernel value towards zero and saturates the hidden units of the MLP.

Full-size 8-bit models were as robust as their baselines or more so. LR-int8 reached 86.0% under σ = 2, against 71.7% for fp32 LR, and kNN-u8 reached 96.6% under σ = 4, against 87.8%. We traced this to the input quantizer of (1), which clips every value to the range seen in training.

Clipping the inputs of the fp32 models to the same range, with no other change, gave the same accuracies to one decimal place (LR 86.0% and 70.1% under σ =

2 and 4; kNN 98.3% and 96.6%), confirming the explanation. Clipping helped the MLP only slightly (28.0% to 33.7%) and did not help the SVM at all, because the clipped noise in near-constant pixels is still large after standardisation. Structural reduction, in contrast, did reduce robustness: under σ = 4 RF20d8 reached 78.3% against 87.3% for the full forest, and changed 19.3% of the full forest's predictions, compared with 0.8% for RF100-u8.

D. Accuracy–memory trade-off (RQ4)

Fig. 4 plots accuracy against compiled flash for DIGITS, and Table VII lists the Pareto-optimal configurations, those for which no smaller configuration is more accurate. On WINE and WDBC the front contains only logistic regression models, which need 214–288 bytes of flash and reach 97.5–98.9% accuracy; no larger model was more accurate on average.

On DIGITS the front runs from LR-int4 (0.6 KB, 96.3%) through LR-int8 and MLP-int8 to Proto10-u8 (6.5 KB, 98.3%), and then to SVM-u8 and kNN-u8, which add at most 0.35 pp for 11 to 14 times more flash and do not fit the 8-bit device. Proto10-u8 is thus the most accurate configuration that fits the ATmega328P for DIGITS, at about 1/55 of the flash of the original kNN.

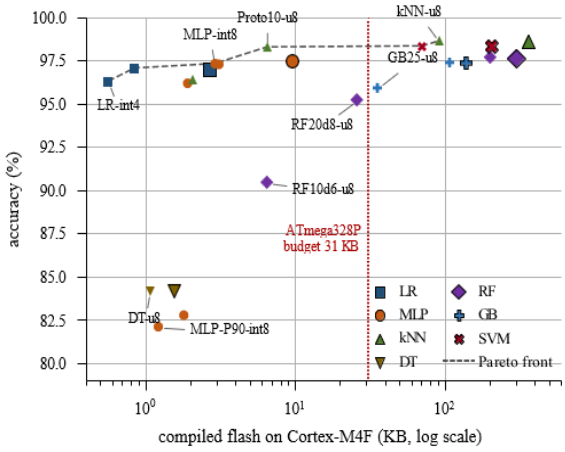

Fig. 4. Accuracy versus compiled flash on the Cortex-M4F for DIGITS (mean over 15 folds). Large outlined markers are fp32 baselines. The dashed line joins the Pareto-optimal configurations; the dotted line is the 31 KB flash budget of the ATmega328P.

Table VII Pareto-optimal configurations (accuracy vs. M4 flash)

Configuration	Acc. (%)	Flash (KB)	Fits AVR
DIGITS			
LR-int4	96.3	0.6	✓
LR-int8	97.0	0.8	✓
MLP-int8	97.3	2.9	✓
Proto10-u8	98.3	6.5	✓
SVM-u8	98.3	71	✗
kNN-u8	98.6	92	✗
WINE			
LR-int8	98.7	0.2	✓
LR-fp32	98.9	0.3	✓
WDBC			
LR-int4	97.5	0.2	✓
LR-int8	97.9	0.2	✓

Each row is more accurate than every smaller configuration of the same dataset.

V. DISCUSSION

A. Practical guidance

Four lessons follow for engineers who deploy classifiers on microcontrollers.

Always store 8-bit parameters. In our experiments 8-bit weights, prototypes, support vectors and tree thresholds never cost significant accuracy, reduced flash by up to 3.9 times and, through the clipping in the input quantizer, often improved robustness. Training trees directly on 8-bit inputs also removes all floating-point code from tree ensembles, which matters on processors without a floating-point unit.

Go below 8 bits only after testing. Four-bit weights and 75% pruning caused no significant loss on WINE and WDBC, but on DIGITS they cost 14.7 pp and 1.3 pp, and 90% pruning reduced accuracy on every dataset. The safe bit-width and sparsity must be measured for each task, as in Fig. 2; there is no universal setting.

Prefer structural reduction for large models. When a model is too large, prototypes and smaller ensembles reduced its size by a further 3 to 31 times beyond 8-bit storage, at a cost of 0.4–2.4 pp in our best cases (Proto10, GB25 and RF20d8 on DIGITS). They should, however, be re-checked for robustness,

because they lost more accuracy under noise than the full models.

Check robustness and input scaling, not only accuracy. The most accurate models on clean data, the SVM and the MLP, were the most fragile under sensor noise, because standardisation amplified noise in near-constant inputs. Clipping inputs to their training range is a free safeguard and is implied by 8-bit quantization. Table VIII summarises these recommendations by device class.

Table VIII Deployment guidance derived from the results

Device class	Recommended	Use with care
8-bit, ≤ 32 KB flash, ≤ 2 KB RAM	LR-int8, MLP-int8, kNN prototypes (u8), small u8 forests	int4 and pruning above50% (task-dependent); SVM (RAM)
32-bit, ~1 MB flash, FPU	Any 8-bit model; full u8 forests and kNN if latency allows	SVM and MLP if inputs are noisy
Any device	Clip inputs to the training range; verify C output against the reference	Reporting parameter counts instead of compiled size

B. Threats to validity

The datasets are small and come from single sources, so the absolute accuracies should not be generalised; the relative effects of compression are the main result. Hyperparameters were fixed rather than tuned, which keeps the comparison fair but may understate what each family can achieve.

Flash and stack were measured on object files produced by one compiler, Clang 18; other compilers and optimisation levels give somewhat different code sizes, although the parameter arrays, which dominate the totals, do not change. Run-time library code and the bootloader were not included in the flash figures; the 1 KB reserve may be too small when software floating point is needed, so models close to the budget should be checked on the board. Instruction counts and latency were measured on an x86-64 host and indicate relative cost only. Energy was not measured.

C. Limitations

- No measurements were taken on physical boards; on-device latency and energy remain to be measured.
- Only post-training quantization was used; quantization-aware training could recover some of the loss at 4 bits or below [10], [23].
- The corruptions were synthetic and applied only to DIGITS.
- Deep convolutional networks, which dominate image tasks on larger microcontrollers [11], were outside the scope of this study.

VI. CONCLUSION AND FUTURE WORK

This paper measured how quantization, pruning and structural reduction affect the accuracy, compiled size and robustness of seven classifier families on two microcontroller targets. Across 72 compiled and verified models, 8-bit storage never cost significant accuracy, reduced flash by up to 3.9 times and often improved robustness, because the input quantizer clips out-of-range values. Four-bit weights were safe on the small tabular tasks but not on the 64-input image task, and pruning 90% of the weights reduced accuracy on every dataset.

For the largest models, replacing the training set by class prototypes or shrinking an ensemble gave the largest savings, and a 10-prototype nearest-neighbour model reached 98.3% on handwritten digits in 6.5 KB of flash, within the budget of an 8-bit Arduino-class device.

The study also showed that the models most accurate on clean data can be the most fragile under sensor noise. We recommend that TinyML studies report compiled size on a named target, verify the deployed code against the trained model, and test robustness alongside accuracy.

Future work will measure latency and energy on physical ATmega328P and nRF52840 boards, add quantization-aware training and knowledge distillation [45] for sub-8-bit models, extend the study to real sensor datasets from agriculture and healthcare, and include small convolutional networks.

DECLARATIONS

Funding. This research received no specific grant from any funding agency.

Conflict of interest. The author declares no conflict of interest.

Data and code availability.

All datasets are public and distributed with scikit-learn. The code that reproduces every table and figure is available from the corresponding author on request.

Use of generative AI.

During the preparation of this work the author used Claude (Anthropic) to assist with writing analysis code and drafting the manuscript. The author reviewed and edited the content and takes full responsibility for the published article.

REFERENCES

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[2] P. Warden and D. Situnayake, TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers. Sebastopol, CA, USA: O'Reilly Media, 2019.

[3] P. P. Ray, "A review on TinyML: State-of-the-art and prospects," Journal of King Saud University – Computer and Information Sciences, vol. 34, no. 4, pp. 1595–1623, 2022.

[4] S. S. Saha, S. S. Sandha, and M. Srivastava, "Machine learning for microcontroller-class hardware: A review," IEEE Sensors Journal, vol. 22, no. 22, pp. 21362–21390, 2022.

[5] R. Sanchez-Iborra and A. F. Skarmeta, "TinyML-enabled frugal smart objects: Challenges and opportunities," IEEE Circuits and Systems Magazine, vol. 20, no. 3, pp. 4–18, 2020.

[6] Microchip Technology Inc., ATmega48A/PA/88A/PA/168A/PA/328/P megaAVR Data Sheet, DS40002061B. [Online]. Available: https://www.microchip.com/en-us/product/atmega328p. [Accessed: Oct. 21, 2026].

[7] Nordic Semiconductor, "nRF52840 system on chip," product page and product specification. [Online]. Available: https://www.nordicsemi.com/Products/nRF52840. [Accessed: Oct. 21, 2026].

[8] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," in Proc. 4th Int. Conf. Learning Representations (ICLR), 2016.

[9] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," arXiv:2103.13630, 2021.

[10] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A white paper on neural network quantization," arXiv:2106.08295, 2021.

[11] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, "MCUNet: Tiny deep learning on IoT devices," in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 11711–11722.

[12] C. Banbury et al., "MLPerf Tiny benchmark," in Proc. Neural Information Processing Systems Track on Datasets and Benchmarks, vol. 1, 2021.

[13] S. Hooker, A. Courville, G. Clark, Y. Dauphin, and A. Frome, "What do compress deep neural networks forget?" arXiv:1911.05248, 2019.

[14] R. David et al., "TensorFlow Lite Micro: Embedded machine learning for TinyML systems," in Proc. Machine Learning and Systems (MLSys), vol. 3, 2021, pp. 800–811.

[15] A. Kumar, S. Goyal, and M. Varma, "Resource-efficient machine learning in 2 KB RAM for the Internet of Things," in Proc. 34th Int. Conf. Machine Learning, PMLR, vol. 70, 2017, pp. 1935–1944.

[16] C. Gupta et al., "ProtoNN: Compressed and accurate kNN for resource-scarce devices," in Proc. 34th Int. Conf. Machine Learning, PMLR, vol. 70, 2017, pp. 1331–1340.

[17] P. E. Hart, "The condensed nearest neighbor rule," IEEE Transactions on Information Theory, vol. 14, no. 3, pp. 515–516, 1968.

[18] S. P. Lloyd, "Least squares quantization in PCM," IEEE Transactions on Information Theory, vol. 28, no. 2, pp. 129–137, 1982.

[19] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, Classification and Regression Trees. Belmont, CA, USA: Wadsworth, 1984.

[20] L. Breiman, "Random forests," Machine

Learning, vol. 45, no. 1, pp. 5–32, 2001.
[21]J. H. Friedman, “Greedy function approximation: A gradient boosting machine,” The Annals of Statistics, vol. 29, no. 5, pp. 1189–1232, 2001.
[22]G. Ke et al., “LightGBM: A highly efficient gradient boosting decision tree,” in Advances in Neural Information Processing Systems, vol. 30, 2017, pp. 3146–3154.
[23]B. Jacob et al., “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR), 2018, pp. 2704–2713.
[24]R. Krishnamoorthi, “Quantizing deep convolutional networks for efficient inference: A whitepaper,” arXiv:1806.08342, 2018.
[25]S. Han, J. Pool, J. Tran, and W. J. Dally, “Learning both weights and connections for efficient neural networks,” in Advances in Neural Information Processing Systems, vol. 28, 2015, pp. 1135–1143.
[26]D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag, “What is the state of neural network pruning?” in Proc. Machine Learning and Systems (MLSys), vol. 2, 2020, pp. 129–146.
[27]J. Frankle and M. Carbin, “The lottery ticket hypothesis: Finding sparse, trainable neural networks,” in Proc. 7th Int. Conf. Learning Representations (ICLR), 2019.
[28]M. Horowitz, “1.1 Computing’s energy problem (and what we can do about it),” in IEEE Int. Solid-State Circuits Conf. (ISSCC) Digest of Technical Papers, 2014, pp. 10–14.
[29]V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, “Efficient processing of deep neural networks: A tutorial and survey,” Proceedings of the IEEE, vol. 105, no. 12, pp. 2295–2329, 2017.
[30]R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, “Green AI,” Communications of the ACM, vol. 63, no. 12, pp. 54–63, 2020.
[31]D. Hendrycks and T. Dietterich, “Benchmarking neural network robustness to common corruptions and perturbations,” in Proc. 7th Int. Conf. Learning Representations (ICLR), 2019.
[32]F. Pedregosa et al., “Scikit-learn: Machine learning in Python,” Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.
[33]E. Alpaydin and C. Kaynak, “Optical recognition of handwritten digits,” UCI Machine Learning Repository, 1998, doi: 10.24432/C50P49.
[34]S. Aeberhard and M. Forina, “Wine,” UCI Machine Learning Repository, 1992, doi: 10.24432/C5PC7J.
[35]W. N. Street, W. H. Wolberg, and O. L. Mangasarian, “Nuclear feature extraction for breast tumor diagnosis,” in Proc. SPIE 1905, Biomedical Image Processing and Biomedical Visualization, 1993, pp. 861–870, doi: 10.1117/12.148698.
[36]D. R. Cox, “The regression analysis of binary sequences,” Journal of the Royal Statistical Society: Series B, vol. 20, no. 2, pp. 215–242, 1958.
[37]D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in Proc. 3rd Int. Conf. Learning Representations (ICLR), 2015.
[38]T. Cover and P. Hart, “Nearest neighbor pattern classification,” IEEE Transactions on Information Theory, vol. 13, no. 1, pp. 21–27, 1967.
[39]C. Cortes and V. Vapnik, “Support-vector networks,” Machine Learning, vol. 20, no. 3, pp. 273–297, 1995.
[40]C.-C. Chang and C.-J. Lin, “LIBSVM: A library for support vector machines,” ACM Transactions on Intelligent Systems and Technology, vol. 2, no. 3, Art. no. 27, 2011.
[41]C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in Proc. Int. Symp. Code Generation and Optimization (CGO), 2004, pp. 75–86.
[42]N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *Proc. 28th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2007, pp. 89–100.
[43]F. Wilcoxon, “Individual comparisons by ranking methods,” *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.
[44]J. Demšar, “Statistical comparisons of classifiers over multiple data sets,” *Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006.
[45]G. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network,” arXiv:1503.02531, 2015.