technologies

MDPI

*Article*

# TinyML for Sustainable Edge Intelligence: Practical Optimization Under Extreme Resource Constraints

**Mohamed Echchidmi * and Anas Bouayad**

Laboratory of Artificial Intelligence, Data Science, and Emerging Systems, Sidi Mohamed Ben Abdellah University, Fez 30050, Morocco

* Correspondence: mohamed.echchidmi@usmba.ac.ma

**Abstract**

Deep learning has emerged as an effective tool for automatic waste classification, supporting cleaner cities and more sustainable recycling systems. Because environmental protection is central to the United Nations Sustainable Development Goals (SDGs), improving the sorting and processing of everyday waste is a practical step toward this broader objective. In many real-world settings, however, waste is still sorted manually, which is slow, labor-intensive, and prone to human error. Although convolutional neural networks (CNNs) can automate this task with high accuracy, many state-of-the-art models remain too large and computationally demanding for low-cost edge devices intended for deployment in homes, schools, and small recycling facilities. In this work, we investigate lightweight waste-classification models suitable for TinyML deployment while preserving competitive accuracy. We first benchmark multiple CNN architectures to establish a strong baseline, then apply complementary compression strategies including quantization, pruning, singular value decomposition (SVD) low-rank approximation, and knowledge distillation. In addition, we evaluate an RL-guided multi-teacher selection benchmark that adaptively chooses one teacher per minibatch during distillation to improve student training stability, achieving up to 85% accuracy with only 0.496 M parameters (FP32 $\approx$ 1.89 MB; INT8 $\approx$ 0.47 MB). Across all experiments, the best accuracy–size trade-off is obtained by combining knowledge distillation with post-training quantization, reducing the model footprint from approximately 16 MB to 281 KB while maintaining 82% accuracy. The resulting model is feasible for deployment on mobile applications and resource-constrained embedded devices based on model size and TensorFlow Lite Micro compatibility.

**Keywords:** knowledge distillation; model compression; waste classification; edge computing; TinyML; sustainability

Check for updates

Academic Editor: Miklas Scholz

Received: 15 February 2026
Revised: 27 March 2026
Accepted: 30 March 2026
Published: 7 April 2026

**Copyright:** © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

## 1. Introduction

Municipal solid waste (MSW) is increasing in both volume and complexity, creating direct environmental and public-health pressures and motivating SDG-aligned improvements in collection, sorting, and recycling. Recent global assessments estimate that the world generates on the order of 2.0–2.1 billion tonnes of MSW per year and project a rise to roughly 3.4–3.8 billion tonnes by 2050 under business-as-usual trends [1,2]. In addition to the growth in volume, a substantial fraction of waste is still managed in environmentally unsafe ways; for example, the World Bank reports that at least 33% of MSW is not managed safely, with open dumping remaining a dominant disposal pathway in lower-income settings [1]. Global material recovery also remains limited, with only a minority

of MSW recovered through recycling and composting according to recent international assessments [1,2]. Together, these figures highlight that improving MSW management is not only a local operational issue, but also a global sustainability challenge.

A key bottleneck in recycling value chains is sorting: materials must be separated with sufficient purity to enable downstream recovery. In many settings, sorting is still performed manually, which is slow, labor-intensive, and prone to inconsistency and error. In low- and middle-income contexts, recovery is often strongly supported by the informal sector, which performs collection and sorting activities and can achieve non-trivial recycling rates even when municipal systems are limited [3,4]. This reality is also documented in country programs: in Morocco, for instance, the World Bank notes that the informal sector constitutes a large workforce involved in collection, sorting, and recycling, and that scaling up sorting-center initiatives and organizing waste pickers remains an ongoing challenge [5]. These observations motivate practical tools that can assist sorting in resource-constrained environments and reduce dependence on purely manual workflows.

Computer vision has therefore become an important enabling technology for waste recognition and automated sorting. Deep learning, particularly convolutional neural networks (CNNs), has substantially advanced automated waste classification by enabling the reliable recognition of recyclable materials from images [6,7]. Accurate recognition can improve separation at the source and the consistency of recovered material streams, but many high-performing CNNs remain computationally and memory intensive. Even mobile-oriented families such as MobileNet and EfficientNet often contain millions of parameters and require non-trivial compute and memory, which can be difficult to accommodate on low-cost embedded and mobile platforms where compute, memory, and energy budgets are tightly limited [8,9]. This constraint is especially relevant for embedded and TinyML-style targets, where available flash memory may be well below a few megabytes and model storage budgets are frequently at the sub-megabyte level [10]. Enabling sustainable edge intelligence thus requires models that are not only accurate, but also compact and efficient enough for practical deployment at the "extreme edge" (e.g., microcontrollers).

From a modeling perspective, there is a constant tension between accuracy and compactness. Larger backbones can deliver strong accuracy on waste-classification datasets (e.g., garbage-classification), but they are often over-parameterized for resource-constrained use cases. Simply shrinking depth or width typically leads to sharp drops in accuracy, particularly when training on relatively small datasets. This motivates model optimization and compression strategies that preserve predictive quality while reducing memory footprint and computation. Knowledge distillation (KD) is especially attractive in this context because it transfers the predictive behavior of a high-capacity teacher into a much smaller student network without inheriting the teacher's full cost. However, KD alone is rarely sufficient for microcontroller-class deployments, where aggressive reductions are often required; practical TinyML systems frequently require combining KD with additional techniques such as quantization, pruning, and low-rank approximations.

Although prior waste-classification studies have explored lightweight architectures and compression, several methodological limitations often remain: (i) efficiency is not consistently reported using a unified set of objective indicators (e.g., parameter count, FP32 footprint as a design reference, INT8 deployment size, and on-disk checkpoint size), which makes comparisons across methods difficult; (ii) compression techniques are frequently evaluated in isolation rather than through a systematic, multi-strategy comparison under a shared training protocol; and (iii) deployment constraints at the extreme edge (sub-megabyte storage, integer inference) are not always reflected in the final model selection criteria. Consequently, the main gap addressed in this study is to provide a practical and reproducible evaluation of the accuracy–efficiency trade-off for TinyML-ready waste

classification by benchmarking multiple compression strategies under consistent experimental conditions.

Therefore, in this work, we present a systematic optimization pipeline for resource-constrained waste classification. We first benchmark multiple CNN baselines to establish a strong reference point for accuracy and cost. We then apply complementary compression strategies—knowledge distillation, post-training quantization, pruning, and SVD-based low-rank approximation—and evaluate them under a unified reporting protocol. In addition, we introduce a reinforcement learning-guided distillation approach that adapts teacher selection during training as a distillation benchmark. Finally, we compare all candidates using objective efficiency indicators and identify the most suitable model under extreme edge constraints, targeting deployment on both mobile devices and microcontrollers.

The model design follows a two-stage constraint. First, we enforce a training-time constraint on the FP32 student model to remain below 500 KB, ensuring a compact architecture. Second, the final deployment target is defined in INT8 precision, with models not exceeding approximately 300 KB flash memory. All deployment conclusions are based on INT8 size. Our main contributions are:

- A practical, end-to-end TinyML optimization pipeline for waste classification that systematically benchmarks multiple compression strategies (KD, quantization, pruning, and SVD low-rank approximation) under a shared protocol.
- A reinforcement learning-guided multi-teacher distillation benchmark that adaptively guides teacher selection during training, serving as an exploratory benchmark for adaptive teacher selection for accuracy retention at a fixed student footprint.
- A unified, quantitative evaluation of accuracy–efficiency trade-offs using objective indicators (top-1 accuracy, parameter count, FP32/INT8 footprint, and checkpoint size), enabling reproducible comparisons across methods.
- A deployment-oriented model selection yielding compact INT8 models (down to ∼300 KB) suitable for extreme-edge deployment on mobile and microcontroller-class devices.

## 2. Related Work

### *2.1. CNNs for Waste Classification*

Public datasets such as the Kaggle `garbage-classification` dataset [11] have enabled supervised learning approaches for recognizing recyclable and non-recyclable materials from images. Early studies typically relied on relatively shallow CNNs or transfer learning from ImageNet pre-trained backbones [12,13]. As the field progressed, deeper architectures such as VGG and ResNet were adopted, improving accuracy but increasing parameter count and computational cost [14,15]. While mobile-oriented families (e.g., MobileNet, EfficientNet) reduce this gap, many variants still remain costly for deployment on low-cost edge devices where memory, compute, and energy are tightly constrained.

To address these limitations, recent work has explored lightweight backbones, depthwise-separable convolutions, and compact CNN designs specifically for waste classification. Examples include WasteNet [16] and compressed MobileNet variants [17], which demonstrate that smaller models can reach near real-time performance with reduced resource usage. However, achieving a robust accuracy–efficiency balance under extreme edge constraints (sub-megabyte storage, CPU-only inference, and tight energy budgets) remains challenging, especially when datasets are limited in scale and diversity. Our study extends this line of research by focusing on TinyML-ready waste recognition and by systematically evaluating multiple optimization strategies under a unified protocol, emphasizing the accuracy–size trade-offs required for practical deployment on microcontrollers and low-end mobile devices.

To make the distinction from prior efficiency-oriented studies explicit, Table 1 summarizes representative lightweight/edge-oriented waste-classification works in terms of datasets, backbones, efficiency strategies, and reported accuracy/complexity. Existing studies commonly propose (or fine-tune) a single compact architecture and report performance using heterogeneous indicators, which complicates reproducible comparisons. In contrast, our work benchmarks several compression strategies—including knowledge distillation, quantization, pruning, and SVD low-rank approximation—under consistent training and evaluation settings, enabling a controlled comparison of the accuracy–efficiency trade-off for TinyML deployment.

**Table 1.** Representative lightweight/efficiency-oriented waste-classification studies (NR: not reported).

| Study | Dataset | Backbone | Efficiency | Key Result(s) | Diff. vs. Ours |
|---|---|---|---|---|---|
| Bircanoglu et al. (2018) [18] | study-specific | DenseNet121; RecycleNet | Arch. optimization | ∼7M→∼3M params; up to 95% acc. | No unified multi-strategy benchmark |
| White et al. (2020) [19] | garbage-classification (10) | WasteNet | Edge CNN design | 97% acc.; Jetson Nano | No KD+compression pipeline; limited efficiency reporting |
| Zheng et al. (2023) [20] | WasteImage | Focus-RCNet | Lightweight net; FLOPs | 92% acc.; 418.8M FLOPs | No distillation/quantization comparison |
| Yong et al. (2023) [21] | domestic (4) | MobileNetV2 | Lightweight TL | 82.92% acc. | No systematic compression study |
| Majchrowska et al. (2022) [22] | classify-waste (7) | EffNet-B2 (cls) | Pipeline/training | ∼75% cls. acc. | Different benchmark; not TinyML-focused |

### 2.2. Knowledge Distillation

Knowledge distillation (KD) was introduced by Hinton et al. [23] as a teacher–student framework in which a smaller model learns not only from hard labels but also from the softened output probabilities of a larger, pre-trained teacher. Subsequent work extended KD in several directions, including feature-based distillation [24], attention transfer [25], and broader surveys that position KD as a practical tool for compression and edge deployment [26]. Multi-teacher and mutual learning variants have also been explored [27,28], but in this study we primarily adopt a single-teacher setup to keep the pipeline simple, reproducible, and compatible with resource-limited training and deployment scenarios.

In embedded vision settings, KD enables compact architectures to inherit much of the teacher's generalization behavior without replicating its full computational footprint. This makes KD a natural companion to additional compression steps such as quantization and pruning, which can otherwise introduce non-trivial accuracy drops when applied aggressively. In our work, KD serves as a central mechanism for preserving accuracy while shrinking the model, and it is evaluated both as a standalone strategy and in combination with other optimization methods (e.g., post-training INT8 quantization) to meet extreme edge storage constraints.

### 2.3. Model Compression

Model compression techniques such as pruning and quantization aim to reduce storage and computation while preserving accuracy. Pruning removes redundant weights or channels [29,30], whereas quantization maps floating-point parameters to low-bit integer formats (e.g., INT8) [31,32]. Low-rank approximation methods (e.g., SVD) further reduce

effective capacity by factorizing weight matrices and constraining rank, offering a complementary pathway to reduce memory and compute. Prior work has also shown that KD can be combined with compression to mitigate accuracy loss caused by aggressive reductions [33]. Building on these insights, our study benchmarks KD, pruning, quantization, and SVD-based low-rank approximation under a shared protocol, and also evaluates combined pipelines (e.g., KD + quantization) for maximizing accuracy within sub-megabyte deployment budgets.

### *2.4. Edge and Mobile AI*

Edge-AI frameworks and mobile-oriented CNNs such as MobileNetV3 [34], MobileNetV4 [35], EfficientNet-Lite [36], and ShuffleNet [37] demonstrate that careful architectural design can significantly reduce latency and memory usage on resource-constrained devices. Ultra-compact solutions such as MCUNet [38] further illustrate the potential of specialized models for microcontroller-class hardware. Our work aligns with this edge perspective, but rather than proposing a new architecture from scratch, we focus on practical optimization: starting from competitive baselines and applying systematic compression (KD, quantization, pruning, and low-rank approximation) to produce models suitable for deployment on both mobile devices and microcontrollers. This emphasis on unified reporting and deployment-oriented evaluation enables a clearer understanding of which optimization combinations provide the best accuracy–size trade-off in TinyML waste-classification settings.

## 3. Mathematical Framework

### *3.1. Problem Formulation*

Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ denote the Kaggle garbage-classification dataset [11] with $N$ labeled RGB images and $C = 10$ classes: battery, biological, cardboard, clothes, glass, metal, paper, plastic, shoes, and trash. Each sample $x_i \in \mathbb{R}^{H\times W\times 3}$ is mapped to logits $z_i \in \mathbb{R}^C$ by a neural network $f_\theta$ with parameters $\theta$,

$$f_\theta : \mathbb{R}^{H\times W\times 3} \to \mathbb{R}^C, \qquad z_i = f_\theta(x_i), \tag{1}$$

and the predicted class distribution is

$$p_{\theta,i} = \mathrm{softmax}(z_i). \tag{2}$$

Our objective is to learn TinyML-ready models that maintain competitive classification performance while satisfying strict deployment constraints on memory, compute, and latency. Concretely, we consider a high-capacity teacher model $f_T$ (EfficientNetB0, evaluated at $224 \times 224$) with parameters $\theta_T$ and a set of optimized candidates $\{f^{(m)}\}_{m\in\mathcal{M}}$ obtained through compression/optimization operators $\mathcal{C}_m$ (PTQ/QAT quantization, pruning, SVD low-rank factorization, and knowledge distillation),

$$f^{(m)} = \mathcal{C}_m(f_{\theta_0}), \qquad \theta^{(m)} = \mathcal{C}_m(\theta_0), \tag{3}$$

and we select models that maximize accuracy subject to extreme edge constraints (e.g., sub-megabyte storage and real-time inference). For the deployable student, we additionally enforce a training-time FP32 design constraint (<500 KB) to ensure a compact student architecture prior to quantization.

### *3.2. Supervised Learning Objective*

Models are trained with cross-entropy loss with optional label smoothing and (when used) class weighting to mitigate dataset imbalance. Let $p_{i,c}$ be the predicted probability for class $c$. With label smoothing $\varepsilon$, the target distribution $q_i \in \mathbb{R}^C$ is

$$q_{i,c} = \begin{cases} 1-\varepsilon, & c = y_i, \\ \dfrac{\varepsilon}{C-1}, & c \neq y_i, \end{cases} \tag{4}$$

and the supervised loss is

$$\mathcal{L}_{\mathrm{CE}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} q_{i,c}\log p_{i,c}. \tag{5}$$

To handle class imbalance in `garbage-classification`, we use inverse-frequency weights computed from the training split: with $n_c$ training samples in class $c$, $N = \sum_c n_c$, and $K = C$, we define

$$w_c = \frac{N}{K\,\max(1, n_c)}, \tag{6}$$

and apply $w_{y_i}$ inside cross-entropy (weights are optionally re-normalized to have mean 1).

### *3.3. Knowledge Distillation*

Knowledge distillation (KD) transfers information from a strong teacher to a compact student using softened output distributions [23]. Given teacher logits $z_{T,i} = f_T(x_i)$ and student logits $z_{S,i} = f_S(x_i)$, temperature-scaled probabilities for $T > 1$ are

$$p_{T,i}^{(T)} = \mathrm{softmax}\Big(\frac{z_{T,i}}{T}\Big), \qquad p_{S,i}^{(T)} = \mathrm{softmax}\Big(\frac{z_{S,i}}{T}\Big). \tag{7}$$

The distillation loss is the KL divergence between teacher and student soft distributions:

$$\mathcal{L}_{\mathrm{KD}} = \frac{T^2}{N}\sum_{i=1}^{N}\mathrm{KL}\Big(p_{T,i}^{(T)} \,\|\, p_{S,i}^{(T)}\Big), \qquad \mathrm{KL}(a\|b) = \sum_{c=1}^{C} a_c \log\frac{a_c}{b_c}. \tag{8}$$

The total KD objective combines hard-label supervision with soft teacher guidance:

$$\mathcal{L}_{\mathrm{total}} = (1-\alpha)\,\mathcal{L}_{\mathrm{CE}} + \alpha\,\mathcal{L}_{\mathrm{KD}}, \tag{9}$$

where $\alpha \in [0,1]$ trades off supervised learning and distillation. In our deployable pipeline, the teacher is EfficientNetB0 and the student is a depthwise-separable CNN (`TinyDSCNN`) operating at $96 \times 96$; during KD, teacher logits are computed on-the-fly by resizing student inputs to $224 \times 224$.

### *3.4. Compression and Optimization Operators*

To satisfy extreme edge constraints, we consider complementary operators that reduce storage and/or computation.

#### 3.4.1. Pruning

Let $W$ denote a weight tensor and $M \in \{0,1\}^{\mathrm{shape}(W)}$ a binary mask. Pruned weights are $\widetilde{W} = M \odot W$. Magnitude pruning can be expressed as

$$M_j = \mathbb{I}\big(|W_j| \geq \gamma\big), \tag{10}$$

where $\gamma$ is chosen to achieve a target sparsity.

3.4.2. Low-Rank Approximation (SVD)

For a weight matrix $W \in \mathbb{R}^{m \times n}$, SVD yields $W = U\Sigma V^\top$. A rank-$r$ approximation replaces $W$ with

$$W \approx U_r \Sigma_r V_r^\top, \tag{11}$$

reducing effective parameterization and potentially lowering memory and compute (depending on implementation).

3.4.3. Quantization

Quantization maps floating-point parameters to low-bit integer formats. Using an affine scheme, a real-valued weight $w$ and its quantized counterpart $\hat{w}$ are related by

$$w_q = \mathrm{clip}\Big(\Big\lfloor \frac{w}{s} \Big\rceil + z,\ q_{\min}, q_{\max}\Big), \qquad \hat{w} = s\,(w_q - z), \tag{12}$$

where $s$ is a scale, $z$ is a zero-point, and $(q_{\min}, q_{\max})$ define the integer range (e.g., INT8). We evaluate both post-training quantization (PTQ) and quantization-aware training (QAT), and for deployment we additionally export the distilled student to TFLite and apply INT8 post-training quantization using a representative calibration subset.

### *3.5. Reinforcement Learning-Guided Multi-Teacher Distillation (Benchmark)*

We benchmark an adaptive distillation variant in which the student architecture is fixed and a controller selects one teacher from a pool at each minibatch. Let $\mathcal{T} = \{T_1, \ldots, T_K\}$ denote the teacher set and $S$ the student. At minibatch step $t$, the controller observes a state $s_t$ (e.g., current epoch and batch-level teacher uncertainty statistics) and chooses an action $a_t \in \{1, \ldots, K\}$ corresponding to selecting teacher $T_{a_t}$. The student is then updated using a distillation objective

$$\mathcal{L}_t = (1-\alpha)\,\mathcal{L}_{\mathrm{CE}}(S(x_t), y_t) + \alpha\,\tau^2\,\mathrm{KL}\Big(\sigma\Big(\frac{z_t^{T_{a_t}}}{\tau}\Big) \,\|\, \sigma\Big(\frac{z_t^S}{\tau}\Big)\Big), \tag{13}$$

where $z_t^{T_{a_t}}$ and $z_t^S$ are teacher/student logits, $\tau$ is the temperature, and $\alpha$ balances supervision and distillation. The controller is trained to maximize a reward that favors improved distillation dynamics (and optionally stability), e.g.,

$$r_t = -\mathcal{L}_t, \tag{14}$$

and learns a policy $\pi_\phi(a_t \mid s_t)$. We report this as a benchmark and include a comparison against a fixed-teacher KD baseline to assess the effect of adaptive teacher selection under a fixed student footprint.

### *3.6. Model Size Estimation*

Let $P(\theta)$ be the number of trainable parameters. Assuming $b$ bytes per parameter, the parameter-memory footprint is approximated by

$$M(\theta, b) = \frac{P(\theta)\,b}{1024^2}\ \mathrm{MB}. \tag{15}$$

We report FP32 footprints ($b = 4$), estimated INT8 footprints ($b = 1$), and the actual on-disk serialized size (TorchScript checkpoints for PyTorch models and FlatBuffer size for TFLite models). These indicators provide a consistent basis for comparing accuracy–efficiency trade-offs. The FP32 size is used as a design-time reference, while deployment feasibility is determined exclusively by INT8 model size.To ensure fair comparison, FP32 and INT8

footprints are reported as parameter-based estimates, while serialized sizes (TorchScript or TFLite) are reported separately and should not be directly compared across frameworks.

## 4. Choosing the Baseline Model

The objective of this work is to study and compare model optimization strategies—including post-training quantization, pruning, low-rank factorization, and knowledge distillation—under a controlled and reproducible experimental setting. A key requirement is therefore to select a baseline architecture that (i) reaches high classification accuracy on the target dataset, (ii) remains compact enough to make compression gains measurable and meaningful, and (iii) is widely supported by standard deployment toolchains (e.g., TensorFlow Lite/PyTorch quantization flows). In particular, a baseline that is too small may saturate early and limit the achievable accuracy, while a baseline that is too large may be costly to train and may obscure the relative impact of optimization techniques.

### *4.1. Evaluation Protocol and Selection Criteria*

To identify a suitable baseline, we trained and evaluated a diverse set of modern CNN and transformer backbones using the same dataset split (train/validation/test) and a consistent evaluation metric. We report the top-1 test accuracy as a measure of predictive performance and the number of parameters (in millions) as a proxy for memory footprint and computational complexity. While parameter count does not fully determine runtime latency (which also depends on operator types and hardware), it provides a consistent first-order measure of model size and is directly relevant to compression methods such as pruning and low-rank decomposition.

The baseline selection is guided by the following criteria:

- High accuracy: The baseline should provide strong performance to ensure that subsequent compression does not reduce accuracy below an acceptable level.
- Compactness: The baseline should be sufficiently small to support deployment-oriented experiments and to highlight improvements from optimization.
- Optimization friendliness: The architecture should be compatible with standard optimization methods (structured/unstructured pruning, INT8 quantization, and distillation workflows) and have stable, well-understood training behavior.

### *4.2. Architectures Evaluated*

We evaluated eight architectures spanning mobile CNNs, standard CNN baselines, modern convolutional designs, and a transformer-based model:

- ConvNeXtTiny;
- EfficientNetV2B0;
- EfficientNetB0;
- ResNet50;
- EfficientNetV2S;
- MobileNetV3Small;
- MobileNetV2;
- ViT_Small.

### *4.3. Results and Discussion*

Table 2 summarizes the test accuracy and parameter count for all evaluated architectures.

Several observations can be drawn from these results. First, EfficientNetB0 achieves the best accuracy in our evaluation (98.43%) while remaining relatively compact (4.06 M parameters). This makes it a strong baseline when the goal is to study compression and

deployment-oriented optimization, since it provides high performance before applying size-reduction techniques.

**Table 2.** Test accuracy and parameter count of the evaluated architectures.

| Model | Test Acc. (%) | Params (M) |
|---|---|---|
| ConvNeXtTiny | 94.71 | 27.83 |
| EfficientNetV2B0 | 93.51 | 5.93 |
| EfficientNetB0 | 98.43 | 4.06 |
| ResNet50 | 92.99 | 23.61 |
| EfficientNetV2S | 92.77 | 20.34 |
| MobileNetV3Small | 90.98 | 0.94 |
| MobileNetV2 | 88.96 | 2.27 |
| ViT_Small | 59.58 | 3.43 |

Second, ConvNeXtTiny has substantially higher capacity (27.83 M parameters) yet does not provide a corresponding accuracy advantage under our training setup, making it less attractive as a baseline for resource-constrained deployment studies.

Third, larger CNN baselines such as ResNet50 and EfficientNetV2S remain significantly heavier (23.61 M and 20.34 M parameters) while achieving lower test accuracy than EfficientNetB0. This indicates that, for this task, increasing capacity beyond EfficientNetB0 yields diminishing returns under the considered protocol.

Finally, lightweight mobile architectures (MobileNetV3Small and MobileNetV2) are significantly smaller and therefore attractive for edge deployment, but they also exhibit a noticeable drop in accuracy. These architectures remain important candidates for the studentrole in knowledge distillation experiments, but they are less suitable as the main baseline if the goal is to preserve strong performance before applying compression.

The transformer baseline (ViT_Small) under-performs substantially (59.58% accuracy), suggesting that, under the current training setup and dataset size, transformer-based training from scratch is less competitive than pre-trained CNN backbones. For this reason, we focus the remainder of the study on CNN-based baselines and students.

### *4.4. Baseline Choice*

Based on the results above, we select EfficientNetB0 as the baseline architecture for the remainder of this work. It offers an excellent accuracy–size trade-off: 98.43% test accuracy with only 4.06 M parameters. These properties make EfficientNetB0 a practical and well-motivated baseline for systematically studying optimization strategies, where the objective is to reduce memory and compute cost while maintaining high predictive performance.

## 5. Compression Methods

We evaluate multiple compression techniques to reduce the memory footprint and improve the deployability of image-based waste-classification models. Unless otherwise stated, EfficientNetB0 is used as the FP32 baseline/teacher model. For each method, we report test accuracy, serialized model size, and (when applicable) CPU inference latency measured with the same evaluation pipeline.

### *5.1. Dataset*

All experiments were conducted on the Kaggle garbage-classification dataset [11], organized into 10 waste categories: metal, glass, biological, paper, battery, trash, cardboard, shoes, clothes, and plastic. The dataset contains 13,348 RGB images are provided as pre-split folders (train/val/test), which we keep unchanged for reproducibility. The resulting split sizes are 10,676 training images, 1331 validation images, and 1341 test images.

Table 3 reports the per-class distribution. The dataset is imbalanced (e.g., trash is substantially smaller than glass or clothes), which motivates explicit imbalance handling during training.

**Table 3.** Per-class distribution of the Kaggle `garbage-classification` dataset (fixed split).

| Class | Total | Train | Val | Test |
|---|---|---|---|---|
| metal | 993 | 794 | 99 | 100 |
| glass | 1998 | 1598 | 199 | 201 |
| biological | 810 | 648 | 81 | 81 |
| paper | 1380 | 1104 | 138 | 138 |
| battery | 814 | 651 | 81 | 82 |
| trash | 494 | 395 | 49 | 50 |
| cardboard | 1533 | 1226 | 153 | 154 |
| shoes | 1633 | 1306 | 163 | 164 |
| clothes | 1984 | 1587 | 198 | 199 |
| plastic | 1709 | 1367 | 170 | 172 |
| Total | 13,348 | 10,676 | 1331 | 1341 |

Handling Class Imbalance

The class distribution is imbalanced (e.g., trash: 395/49/50 vs. glass: 1598/199/201 for train/val/test), so we mitigate bias toward majority classes by using inverse-frequency class weights in the cross-entropy loss, computed from the training split as $w_c = N/n_c$ where $n_c$ is the number of training images in class $c$ and $N = \sum_c n_c$ is the total training set size.

### *5.2. Evaluation Protocol*

We report: (i) top-1 accuracy on the held-out test set, (ii) serialized model size (TorchScript for PyTorch models and FlatBuffer size for TFLite models), and (iii) CPU latency (ms/image) where applicable.

For quantized models, INT8 evaluation is performed using the corresponding INT8 runtime graph (PyTorch FX INT8 for PTQ/QAT, and TFLite INT8 for deployment). Latency is measured by averaging multiple forward passes after a warm-up phase.

### *5.3. Implementation Details*

#### 5.3.1. Teacher and Baseline

We fine-tune EfficientNetB0 as the main high-accuracy reference model. Images are resized to the network input resolution and normalized using ImageNet statistics.

#### 5.3.2. Compression Methods

Starting from the best FP32 baseline checkpoint, we apply:

- PTQ (FX graph-mode static quantization) with calibration on training images using deterministic preprocessing (no augmentation);
- QAT (FX graph-mode) initialized from the FP32 checkpoint;
- Unstructured pruning (global magnitude pruning);
- Structured channel pruning with short fine-tuning;
- SVD low-rank factorization of eligible layers with short fine-tuning.

#### 5.3.3. Deployable Student (Ours)

To build a microcontroller/mobile-friendly model, we train a compact depthwise-separable student CNN using knowledge distillation from the EfficientNetB0 teacher

(Section 6). The distilled student is then exported and quantized to INT8 for deployment, following the PyTorch → ONNX → TFLite conversion workflow.

### *5.4. Reproducibility*

Algorithm 1 summarizes the steps required to reproduce the experimental results in this paper. The key aspects for exact reproducibility are: (i) using the same dataset version and the same stratified split, (ii) fixing random seeds for Python/NumPy/framework backends, (iii) keeping preprocessing identical across FP32 and INT8 evaluation, and (iv) using a representative calibration subset for PTQ/TFLite INT8 conversion.

**Algorithm 1.** Reproducing the experiments (training, compression, deployment).

**Input:** Kaggle dataset `garbage-classification`; fixed stratified split (train/val/test); teacher $f_T$ (EfficientNetB0); student $f_S$ (depthwise-separable CNN).
**Output:** All model checkpoints and reported metrics (accuracy/size/latency).

**Step 1: Data preparation**:
Download the Kaggle dataset and verify class folders and counts;
Create stratified split lists (train/val/test) and store filenames;
Define preprocessing (resize + ImageNet normalization).

**Step 2: Teacher/baseline training**:
Fine-tune EfficientNetB0 on the training set, select best checkpoint on validation accuracy;
Evaluate FP32 baseline on the test set; export TorchScript checkpoint; record size/latency.

**Step 3: Compression experiments**:
Apply PTQ using a calibration pass on training images; evaluate INT8; record size/latency;
Run QAT fine-tuning from the FP32 checkpoint; convert to INT8; evaluate; record size/latency;
Apply pruning (unstructured/structured) and evaluate; record accuracy and serialized size;
Apply SVD low-rank factorization and fine-tune briefly; evaluate; record accuracy and size.

**Step 4: Deployable student (ours)**:
Train student $f_S$ with KD from teacher $f_T$; select best checkpoint on validation accuracy;
Export student: PyTorch → ONNX → TensorFlow → TFLite;
Perform INT8 TFLite post-training quantization using a representative calibration subset;
Evaluate the INT8 TFLite model on the test set; record final accuracy and model size.

### *5.5. Results*

Table 4 consolidates the main outcomes. While the FP32 EfficientNetB0 baseline achieves the highest accuracy, it is too large for extreme edge targets. Among compression-only approaches, PTQ provides a strong accuracy–size compromise for the baseline model. For the deployable setting, the best accuracy–size trade-off is obtained by knowledge distillation to a depthwise-separable student followed by INT8 quantization, producing a sub-megabyte model suitable for mobile and microcontroller-class deployment.

**Table 4.** Overall results on Kaggle `garbage-classification`.

| Method/Model | Test Acc. (%) | Serialized Size |
|---|---|---|
| EfficientNetB0 (FP32 baseline) | 98.43 | 15.81 MB (TorchScript) |
| PTQ on EfficientNetB0 (INT8) | 94.41 | 4.62 MB (INT8) |
| QAT on EfficientNetB0 (INT8) | 92.32 | 4.62 MB (INT8) |
| Unstructured pruning (40%) | 96.20 | 15.83 MB (TorchScript) |
| Channel pruning (0.2) | 90.38 | 10.37 MB (TorchScript) |
| SVD low-rank ($\rho = 0.60$) | 92.17 | 12.86 MB (TorchScript) |
| RL-guided student (benchmark, FP32) | 85.00 | 1.89 MB (est.) |
| KD student (depthwise-separable, FP32) | 84.49 | 3.00 MB (est.) |
| Ours: KD student + INT8 TFLite | 82.0 | 281 KB (TFLite) |

### *5.6. Quantization*

#### 5.6.1. Post-Training Quantization (PTQ)

We applied post-training static quantization(PTQ) to convert the trained FP32 EfficientNetB0 model into INT8 without additional fine-tuning. PTQ reduces the precision of weights and activations to decrease model size and improve CPU efficiency. We used FX graph-mode quantization with a calibration pass over training data (deterministic preprocessing, no augmentation). Calibration employed a histogram-based activation observer and per-channel weight quantization to obtain stable activation ranges.

#### 5.6.2. Results (Table 5)

PTQ reduced the model from 15.81 MB (FP32) to 4.62 MB (INT8), a 70.77% reduction. Test accuracy decreased from 98.43% to 94.41% (drop of 4.03 pp), while CPU latency improved from 23.73 to 19.44 ms/image (1.22× speedup).

**Table 5.** PTQ results for EfficientNetB0 (FP32 vs. INT8).

| Metric | FP32 | INT8 (PTQ) |
|---|---|---|
| Test accuracy (%) | 98.43 | 94.41 |
| Model size (MB) | 15.81 | 4.62 |
| CPU latency (ms/image) | 23.73 | 19.44 |

#### 5.6.3. Quantization-Aware Training (QAT)

We further evaluated quantization-aware training (QAT), which simulates INT8 quantization effects during fine-tuning through fake-quantization modules. QAT allows the network to adapt to discretization noise and can improve robustness in low-precision inference.

#### 5.6.4. Method

We initialized QAT from the best FP32 EfficientNetB0 checkpoint and enabled FX graph-mode QAT to insert fake-quantization for activations and weights. Fine-tuning was performed in FP32 with a small learning rate. Near the end of training, we froze quantization observers to fix activation ranges and froze BatchNorm running statistics to avoid distribution drift. The model was then converted to a true INT8 graph and evaluated with the same protocol as PTQ.

#### 5.6.5. Results (Table 6)

QAT produced an INT8 model with the same serialized size as PTQ (4.62 MB) and improved latency relative to FP32, but with a larger accuracy drop: test accuracy decreased to 92.32%. CPU latency improved from 26.83 to 21.58 ms/image (1.24× speedup).

**Table 6.** QAT results for EfficientNetB0 (FP32 vs. INT8).

| Metric | FP32 | INT8 (QAT) |
|---|---|---|
| Test accuracy (%) | 98.43 | 92.32 |
| Model size (MB) | 15.81 | 4.62 |
| CPU latency (ms/image) | 26.83 | 21.58 |

#### 5.6.6. PTQ vs. QAT

In our experiments, post-training quantization (PTQ) achieved higher accuracy than quantization-aware training (QAT) (94.41% vs. 92.32%). While this result may appear counterintuitive, we emphasize that it is specific to our experimental configuration and should not be interpreted as a general conclusion.

QAT performance is known to be sensitive to multiple factors, including observer configuration, BatchNorm handling, learning rate schedule, and calibration consistency. In our setup, QAT was performed using a relatively short fine-tuning schedule and default observer settings, which may not fully exploit its potential. Therefore, we interpret the observed PTQ advantage as an artifact of the current configuration rather than evidence of inherent superiority.

### *5.7. Pruning*

#### 5.7.1. Unstructured Weight Pruning

We applied global unstructured magnitude pruning to EfficientNetB0 by setting individual weights to zero based on global $\ell_1$ magnitude. This introduces elementwise sparsity, which may preserve accuracy at moderate sparsity but typically does not reduce dense model size unless sparse storage/runtime support is used.

#### 5.7.2. Method

For pruning ratios $p \in \{0.2, 0.4, 0.6, 0.8\}$, we pruned the Conv2d and linear layers by zeroing the smallest-magnitude $p$ fraction of weights globally. We removed pruning reparameterization to make sparsity permanent and evaluated the model without fine-tuningto isolate the direct effect of pruning.

#### 5.7.3. Results (Table 7)

The model tolerated 20–40%sparsity with limited accuracy loss, but collapsed beyond this range. Importantly, the TorchScript size remained nearly constant (≈15.83 MB) for all pruning ratios, confirming that unstructured pruning alone does not yield storage savings in a dense representation.

**Table 7.** Global unstructured magnitude pruning results for EfficientNetB0.

| Pruning (%) | Sparsity (%) | Val Acc. (%) | Test Acc. (%) | TorchScript (MB) |
|---|---|---|---|---|
| 0 | 0 | 98.50 | 98.43 | 15.81 |
| 20 | 20 | 98.50 | 98.51 | 15.83 |
| 40 | 40 | 95.94 | 96.20 | 15.83 |
| 60 | 60 | 23.07 | 24.91 | 15.83 |
| 80 | 80 | 3.68 | 3.73 | 15.83 |

#### 5.7.4. Structured Pruning (Channel Pruning)

To achieve deployable compression on dense runtimes, we evaluated structured channel pruning, which removes entire channels/filters and therefore reduces tensor shapes and parameter count.

#### 5.7.5. Method

Starting from the FP32 baseline, we applied magnitude-based channel pruning with target channel sparsity $s \in \{0.2, 0.3, 0.4\}$. Pruning was performed iteratively, followed by short fine-tuning to recover accuracy. We report accuracy, remaining parameters, TorchScript size, and CPU latency.

#### 5.7.6. Results (Table 8)

Structured pruning reduced the model size from 15.81 MB to 10.37 MB ($s = 0.2$) and down to 6.14 MB ($s = 0.4$), but incurred a substantial accuracy drop (test accuracy decreased to 90.38% at $s = 0.2$ and 83.00% at $s = 0.4$). Latency gains were modest in our setup.

**Table 8.** Structured channel pruning results for EfficientNetB0 after fine-tuning.

| Ch. Sparsity | Val Acc. (%) | Test Acc. (%) | Params (M) | TorchScript (MB) | CPU (ms/img) |
|---|---|---|---|---|---|
| Baseline | 98.50 | 98.43 | 4.02 | 15.81 | 25.15 |
| 0.2 | 93.16 | 90.38 | 2.60 | 10.37 | 23.04 |
| 0.3 | 87.83 | 85.98 | 2.01 | 8.11 | 23.02 |
| 0.4 | 84.15 | 83.00 | 1.50 | 6.14 | 23.84 |

### *5.8. Low-Rank Approximation via SVD*

We applied truncated *singular value decomposition* (SVD) to factorize selected layers of EfficientNetB0 into low-rank components. For $1 \times 1$ convolutions, kernels were reshaped into matrices and replaced by two sequential $1 \times 1$ convolutions with intermediate rank $r$, providing a tunable compression factor.

#### 5.8.1. Method

We factorized eligible $1 \times 1$ convolutions (`groups = 1`), excluding squeeze-and-excitation layers for stability, and also factorized the classifier layer. For each layer, we retained a fraction $\rho \in \{0.90, 0.75, 0.60, 0.50\}$ of the maximum rank. Models were fine-tuned for three epochs after factorization.

#### 5.8.2. Results (Table 9)

After fine-tuning, moderate ranks preserved high accuracy while reducing size. The configuration $\rho = 0.60$ achieved 92.17% test accuracy with 12.86 MB TorchScript size, while $\rho = 0.50$ reduced the size further to 11.43 MB at 89.71% test accuracy. Latency did not consistently improve due to additional operator overhead.

**Table 9.** Low-rank SVD factorization results for EfficientNetB0 after fine-tuning.

| Model | $\rho$ (Keep) | Val Acc. (%) | Test Acc. (%) | Params (M) | TorchScript (MB) |
|---|---|---|---|---|---|
| SVD low-rank | 0.90 | 97.67 | 97.76 | 4.27 | 17.21 |
| SVD low-rank | 0.75 | 96.99 | 95.45 | 3.70 | 15.04 |
| SVD low-rank | 0.60 | 93.24 | 92.17 | 3.13 | 12.86 |
| SVD low-rank | 0.50 | 90.23 | 89.71 | 2.75 | 11.43 |

### *5.9. Knowledge Distillation to a Compact Student*

Finally, we evaluated the knowledge distillation (KD) to transfer the performance of the EfficientNetB0 teacher to compact student networks. To address the reviewer concern about student choice, we report a controlled student-architecture comparison under the same KD protocol and training setup.

#### 5.9.1. Student Choice and TinyML Motivation

We trained four candidate students with KD from the same EfficientNetB0 teacher: a custom depthwise-separable CNN(ours), MobileNetV3-Small, MobileNetV2, and SqueezeNet1_1. As summarized in Table 10, MobileNetV3-Small and MobileNetV2 achieved the highest test accuracy (94.48% and 94.05%), but with noticeably larger footprints (1.528 M–2.237 M parameters, corresponding to 5.83–8.53 MB in FP32). SqueezeNet1_1 offers a smaller alternative (0.728 M parameters) with 92.48% test accuracy. Our depthwise-separable CNN reaches 87.28% test accuracy with only 0.484 M parameters (FP32 $\approx$ 1.85 MB; estimated INT8 $\approx$ 0.46 MB), representing a 3–5$\times$parameter reduction compared to MobileNetV3-Small/MobileNetV2 and a $\sim$1.5$\times$reduction compared to SqueezeNet1_1. Because the main objective of this work is extreme-edge feasibility—where memory budgets are typically sub-megabyte and INT8 inference is required—we retain the depthwise-separable CNN as the deployable student architecture, prioritizing the most favorable accuracy–size trade-off among the evaluated options and leaving headroom for subsequent INT8 deployment steps.

#### 5.9.2. Method

All students are trained with the same distillation objective combining (i) hard-label cross-entropy and (ii) a KL-divergence term between teacher and student temperature-softened logits. Each student checkpoint is selected by validation accuracy and then evaluated on the held-out test split. Parameter count is reported as a hardware-agnostic size proxy; FP32 and estimated INT8 footprints are derived from parameter count.

#### 5.9.3. Results (Table 10)

Table 10 reports the resulting accuracy and footprint trade-offs across the four student backbones.

**Table 10.** KD student backbone comparison (teacher: EfficientNetB0; same training protocol).

| Student | Val Acc. (%) | Test Acc. (%) | Params (M) | Est. FP32 (MB) | Est. INT8 (MB) | Checkpoint (MB) |
|---|---|---|---|---|---|---|
| MobileNetV3-Small (KD) | 95.42 | 94.48 | 1.528 | 5.83 | 1.46 | 5.95 |
| MobileNetV2 (KD) | 96.40 | 94.05 | 2.237 | 8.53 | 2.13 | 8.75 |
| SqueezeNet1_1 (KD) | 93.56 | 92.48 | 0.728 | 2.78 | 0.69 | 2.79 |
| Depthwise-separable CNN (KD, ours) | 88.38 | 87.28 | 0.484 | 1.85 | 0.46 | 1.90 |

### *5.10. Reinforcement Learning-Guided Multi-Teacher Distillation (Benchmark)*

To improve distillation robustness under teacher disagreement and to provide a reproducible adaptive baseline, we implement a lightweight reinforcement learning (RL) teacher-selection strategy that chooses one teacher per minibatchImportantly, this benchmark evaluates whether adaptive teacher choice can improve student accuracy at a fixed student footprint; RL does not modify the student architecture or size.

#### 5.10.1. Setup (Dataset and Loaders)

All experiments use the fixed Kaggle `garbage-classification` split with $C = 10$ classes. To mitigate class imbalance during student training, we use inverse-frequency sampling via a `WeightedRandomSampler` computed from the training-set class counts. We train with a minibatch size of $B = 64$ using standard shuffled minibatch data loading.

#### 5.10.2. Teachers (Multi-Teacher Pool)

We construct a teacher pool of $K = 3$ ImageNet-initialized networks: (i) EfficientNetB0, (ii) MobileNetV3Small, and (iii) ResNet50. Each teacher replaces its classifier head with a

$C$-class linear layer and is either loaded from a cached checkpoint or quickly adapted to the dataset using a two-stage schedule: (1) head-only training for two epochs, then (2) light fine-tuning for two epochs (unfreezing late blocks). During student training, all teacher parameters are frozen.

#### 5.10.3. Student (Fixed Footprint in This Benchmark)

The student is a very lightweight depthwise-separable CNN (TinyDSCNN-style) designed to be deployable on memory-constrained targets. In the reported benchmark, we use the exact student configuration that yields 0.496 M parameters (Table 11). Because the RL agent selects only the teacher index, the student footprint remains constant throughout RL training; the benchmark isolates the effect of teacher selection on accuracy at this fixed size.

**Table 11.** RL-guided multi-teacher distillation benchmark (teacher selection; fixed student footprint).

| Model | Test Acc. (%) | Params (M) | Est. FP32 (MB) | Est. INT8 (MB) |
|---|---|---|---|---|
| RL-guided student (benchmark) | 85.00 | 0.496 | 1.89 | 0.47 |

#### 5.10.4. Distillation Objective (Optional Confidence Gating)

We use logit-based knowledge distillation with temperature $T = 4$ and mixing coefficient $\alpha = 0.7$:

$$\mathcal{L} = (1 - \alpha)\,\mathrm{CE}(z_S, y) + \alpha\, T^2\,\mathrm{KL}(\log \mathrm{softmax}(z_S/T) \,\|\, \mathrm{softmax}(z_T/T)). \tag{16}$$

Optionally, a coarse confidence gate can revert to a fixed teacher when the selected teacher is globally uncertain on a minibatch; in the reported experiments, this gate is disabled ($\tau = 0$) to keep the policy gradient unbiased.

#### 5.10.5. RL Formulation (Batch-Level Teacher Selection)

At each step, the agent selects one teacher index $a_t \in \{1, \ldots, K\}$ for the entire minibatch. The state summarizes teacher uncertainty and training progress:

$$s_t = \left[\, \bar{c},\ \overline{H},\ \overline{m},\ p \,\right],$$

where $\bar{c}$ is mean teacher confidence, $\overline{H}$ is mean predictive entropy, $\overline{m}$ is mean top-1 vs. top-2 probability margin, and $p$ is normalized epoch progress. The policy $\pi_\phi(a_t \mid s_t)$ is a two-layer MLP (hidden size 64) producing a categorical distribution over teachers.

#### 5.10.6. RL Optimization (REINFORCE with Baseline)

Because the student footprint is fixed in this benchmark, the RL signal optimizes distillation dynamics rather than an explicit accuracy–size trade-off. The per-batch reward is the negative KD loss:

$$r_t = -\mathcal{L}_t. \tag{17}$$

We use REINFORCE with an exponential moving-average baseline $b_t$ (decay $\beta = 0.95$), yielding advantage the $A_t = r_t - b_t$. The agent is updated every batch by minimizing

$$\mathcal{L}_{\text{agent}} = -\log \pi_\phi(a_t \mid s_t)\, A_t,$$

with gradient clipping ($\|\nabla\|_2 \leq 1$). Intuitively, the agent learns to prefer teachers that produce more useful gradients for the student under the same KD objective.

#### 5.10.7. Training Schedule and Hyperparameters

We use a two-phase schedule: (i) warmup for $E_w = 10$ epochs with a fixed EfficientNetB0 teacher to stabilize early KD, followed by (ii) RL teacher selection for $E_{rl} = 25$ epochs. The student uses AdamW (lr $= 5 \times 10^{-4}$, weight decay $10^{-4}$) with cosine annealing over $E_w + E_{rl}$ epochs. The agent uses AdamW (lr $= 3 \times 10^{-4}$). AMP is enabled when CUDA is available. To account for RL variance, results should be reported over multiple random seeds (e.g., three runs) using mean±std, with the best checkpoint selected by validation accuracy in each run.

#### 5.10.8. Training-Time Overhead

Computing the state features as defined above requires teacher uncertainty statistics. Following Algorithm 2, we compute logits for all $K$ teachers on each minibatch (no gradients) to form $\bar{c}$, $\overline{H}$, and $\overline{m}$ before sampling a teacher. This increases training cost relative to single-teacher KD, but it does not affect deployment, since only the distilled student is used at inference time.

**Algorithm 2.** RL-guided multi-teacher distillation (one teacher per minibatch).

**Input:** Dataset split `/content/garbage_split`; teachers $\{f_{T_k}\}_{k=1}^{K}$; student $f_S$; temperature $T$; $\alpha$.
**Output:** Best student checkpoint $f_S^\star$ selected by validation accuracy.

**Warmup (stabilize KD):**
Fix teacher $k \leftarrow$ EfficientNetB0;
Train $f_S$ for $E_w$ epochs with KD loss;
Select best checkpoint by validation accuracy;

**RL teacher selection:**;
Initialize policy $\pi_\phi(a \mid s)$ over $\{1, \ldots, K\}$ and baseline $b \leftarrow 0$;
**for** $e \leftarrow 1$ **to** $E_{rl}$ **do**
  **for** *each minibatch* $(x, y)$ **do**
    Compute teacher logits $\{z_{T_k}\}$ with no gradients;
    Build state $s \leftarrow [\bar{c}, \overline{H}, \overline{m}, p]$;
    Sample teacher $a \sim \pi_\phi(\cdot \mid s)$ and set $z_T \leftarrow z_{T_a}$;
    Compute student logits $z_S \leftarrow f_S(x)$;
    Compute KD loss $\mathcal{L}(z_S, y, z_T; T, \alpha)$;
    Update student parameters to minimize $\mathcal{L}$;
    Set reward $r \leftarrow -\mathcal{L}$ and update baseline $b \leftarrow \beta b + (1 - \beta) r$;
    Update agent (REINFORCE): minimize $-\log \pi_\phi(a \mid s)\,(r - b)$;
  **end**
  Evaluate val/test accuracy; save best $f_S^\star$ by val accuracy;
**end**

#### 5.10.9. Teacher-Selection Baseline Comparison

To provide a reference point for adaptive teacher selection, we compare the RL-guided strategy to a standard fixed-teacher KD baseline (EfficientNetB0) under the same student configuration (0.496 M parameters) and training budget. The fixed-teacher baseline achieves approximately 84% test accuracy, while RL-guided selection reaches 85% (Table 11). This indicates a modest improvement from adaptive teacher selection. However, the gain remains limited relative to the additional training cost, and therefore RL should be interpreted as a lightweight exploratory enhancement rather than a definitive improvement over all multi-teacher strategies.

#### 5.10.10. Artifacts and Reproducibility

The script saves (i) the best student checkpoint (selected by validation accuracy) and (ii) a TorchScript export. To reproduce results, use the same folder split, augmentations, and teacher adaptation schedule (2 + 2 epochs). Report teacher checkpoints used, student configuration, and seed list.

5.10.11. Training-Time Cost–Benefit Analysis of RL Teacher Selection

The main overhead of the RL controller arises from computing logits for all $K$ teachers at each minibatch to form the policy state. While the warmup phase matches standard single-teacher KD, the RL phase requires $K$ teacher forward passes (no gradients) in addition to the student update, leading to a training cost that scales approximately linearly with $K$. In contrast, random-teacher selection has the same cost as single-teacher KD, and uniform averaging requires $K$ teacher passes but no policy updates.

As summarized in Table 12, RL-based selection provides a modest accuracy improvement under a fixed student footprint, at the expense of increased training compute. This overhead is limited to training; deployment cost remains unchanged since only the distilled student is used. The RL approach is therefore most appropriate when teacher diversity is significant and training resources allow additional computation, while simpler baselines remain effective under tighter training budgets.

**Table 12.** Training-phase cost–benefit summary for multi-teacher distillation ($K$ teachers). $\mathcal{C}_T$ and $\mathcal{C}_S$ denote per-batch forward cost of a teacher and the student; $\mathcal{C}_\pi$ is the (small) policy-network update cost.

| Strategy | Teacher Fwd/Batch | Extra Update | Training Cost (Approx.) |
|---|---|---|---|
| Fixed-teacher KD | 1 | student | $\mathcal{C}_S + \mathcal{C}_T$ |
| Random-teacher KD | 1 | student | $\mathcal{C}_S + \mathcal{C}_T$ |
| Uniform averaging (logit mean) | $K$ | student | $\mathcal{C}_S + K\mathcal{C}_T$ |
| RL teacher selection (ours) | $K$ | student + agent | $\mathcal{C}_S + K\mathcal{C}_T + \mathcal{C}_\pi$ |

**Interpretation:** Random selection provides diversification at the same cost as single-teacher KD (one teacher forward pass per batch). Uniform averaging and RL both require computing all $K$ teacher logits per batch, but RL adds a small policy update; therefore, RL is only cost-effective when it provides a consistent accuracy gain over averaging/random selection, while deployment cost remains unchanged because only the distilled student is used at inference time.

## 6. Designing a Deployable Model

This section describes an end-to-end pipeline for building a lightweight waste-classification model that targets resource-constrained devices and real-time inference. Guided by our experimental findings, we adopt a teacher–student approach for extreme-edge deployment: a high-capacity EfficientNetB0 teacher is trained at 224 × 224 resolution, and its knowledge is distilled into a compact depthwise-separable student (`TinyDSCNN`) trained at 96 × 96 under a training-time FP32 design constraint of <500 KB, followed by INT8 deployment optimization. Importantly, this deployable student is not the same model used in the previous section; it is a further reduced variant designed to enable additional compression and meet tighter memory constraints. After distillation, the student is exported and prepared for deployment via INT8 quantization. Because training is performed in PyTorch, we use an interoperability workflow with ONNX as an intermediate representation to produce a deployment-ready TensorFlow Lite (TFLite) model compatible with mobile and embedded runtimes. The complete procedure is summarized in Algorithm 3.

**Algorithm 3.** Pipeline for designing a deployable TinyML model (KD + INT8 deployment).

**Input:** Dataset $\mathcal{D}$; teacher $f_T$ (EfficientNetB0); student $f_S$ (`TinyDSCNN`); temperature $T$; mixing coefficient $\alpha$; teacher epochs $E_T$; student epochs $E_S$; calibration set $\mathcal{D}_{\text{cal}}$.
**Output:** Deployment-ready INT8 TFLite model $f_S^{\text{INT8}}$.

**Teacher training:**;
Initialize $f_T$ with ImageNet weights and replace classifier head for $C$ classes;
Train $f_T$ on $\mathcal{D}$ for $E_T$ epochs using cross-entropy;
Select and save teacher checkpoint $\theta_T^\star$ by validation accuracy;

**Student distillation:**;
Initialize student $f_S$ under the FP32 design constraint (<500 KB);
**for** $e \leftarrow 1$ **to** $E_S$ **do**
  Sample minibatch $(x, y)$ from $\mathcal{D}$;
  Compute teacher logits $z_T \leftarrow f_T(x; \theta_T^\star)$ (no gradient);
  Compute student logits $z_S \leftarrow f_S(x; \theta_S)$;
  $\mathcal{L}_{\text{CE}} \leftarrow \text{CE}(z_S, y)$;
  $\mathcal{L}_{\text{KD}} \leftarrow T^2\,\text{KL}(\log\text{softmax}(z_S/T) \,\|\, \text{softmax}(z_T/T))$;
  Update $\theta_S$ by minimizing $\mathcal{L} = (1-\alpha)\mathcal{L}_{\text{CE}} + \alpha\mathcal{L}_{\text{KD}}$;
**end**
Save best student checkpoint $\theta_S^\star$;

**Export and deployment preparation:**;
Export $f_S(\cdot; \theta_S^\star)$ to ONNX;
Convert ONNX → TensorFlow;
Convert TensorFlow → TFLite and apply INT8 post-training quantization using $\mathcal{D}_{\text{cal}}$;
Return $f_S^{\text{INT8}}$;

### *6.1. Experimental Setup*

All experiments use the Kaggle `garbage-classification` dataset (10 classes) [11] organized as `/content/garbage_split/{train,val,test}/{class}/...`. Training is implemented in PyTorch and run on GPU when available (AMP enabled). We fix random seeds (`SEED = 42`) and keep the dataset split unchanged. The teacher is trained/evaluated at $224 \times 224$, while the deployable student is trained/evaluated at $96 \times 96$ to reduce compute and activation memory. The best checkpoint is selected by validation accuracy with early stopping (patience $= 10$), then exported for reporting and conversion.

### *6.2. Training Configuration*

#### 6.2.1. Data Preprocessing and Augmentation

Student training uses random resized crops (scale 0.70–1.00), horizontal flips, mild rotations, and `ColorJitter`, followed by ImageNet normalization. Validation and test use deterministic resize and center crop at $96 \times 96$ with the same normalization. During distillation, teacher inputs are produced on-the-fly by resizing the normalized student batch from $96 \times 96$ to $224 \times 224$ (bilinear interpolation), avoiding parallel dataloaders.

#### 6.2.2. Class Imbalance Handling

We apply inverse-frequency class weights computed from training-folder counts. Let $n_c$ be the number of training images in class $c$, $N = \sum_c n_c$, and $K = 10$. We use

$$w_c = \frac{N}{K \cdot \max(1, n_c)},$$

and normalize weights to have a mean of one before passing them to cross-entropy. This weighting is applied to the hard-label loss term used in training.

#### 6.2.3. Teacher Model and Checkpoint Policy

The teacher is EfficientNetB0 initialized with ImageNet weights and adapted to 10 classes by replacing the classifier head. If a saved teacher checkpoint is available, it

is loaded using a robust state-dictionary extractor; otherwise, a short fallback training is performed (classifier head training followed by light fine-tuning), selecting the best checkpoint by validation accuracy.

#### 6.2.4. Student Architecture and Memory Budget

The student `TinyDSCNN` is a depthwise-separable CNN with a two-layer stem, residual depthwise-separable blocks (SiLU), and lightweight SE attention. The architecture is constrained during training to remain under a 500 KB FP32 design limit, prior to INT8 deployment. The training script automatically selects the largest width that satisfies:

$$P \leq \left\lfloor \frac{500 \times 1024}{4} \right\rfloor,$$

where $P$ is the number of trainable parameters. The selected configuration is stored in the checkpoint for exact reconstruction at export time.

#### 6.2.5. Knowledge Distillation Objective

The student is trained using logit-based KD:

$$\mathcal{L} = (1-\alpha)\,\mathcal{L}_{\mathrm{CE}}(z_S, y) + \alpha\, T^2\, \mathrm{KL}(\log \mathrm{softmax}(z_S/T) \,\|\, \mathrm{softmax}(z_T/T)),$$

with $\alpha = 0.75$, $T = 4.0$, and label smoothing $\varepsilon = 0.05$ in the cross-entropy term. The teacher runs in `no_grad` mode during KD.

#### 6.2.6. Optimization and Early Stopping

We train the student for up to $E = 60$ epochs using AdamW (learning rate $2 \times 10^{-3}$, weight decay $10^{-4}$) with cosine annealing. Gradient clipping is applied ($\|\nabla\|_2 \leq 5$). Early stopping is triggered if validation accuracy does not improve for 10 consecutive epochs.

### *6.3. Model Export, Quantization, and Figures*

After training, the best student checkpoint is reloaded and exported to TorchScript using `torch.jit.trace`. For deployment, we convert the model through PyTorch → ONNX → TensorFlow → TFLite and apply INT8 post-training quantization using a representative calibration set at $96 \times 96$.

### *6.4. Deployed Student Metrics (Validation and Test)*

To provide a more complete picture of deployment performance under class imbalance, we report additional metrics for the distilled and INT8-deployed student. Overall validation metrics are summarized in Table 13, and overall test metrics are given in Table 14. Per-class precision/recall/F1 and support are reported for validation and test in Tables 15 and 16, respectively.

**Table 13.** Deployed student (INT8) overall metrics on the validation split.

| Metric | Value |
|---|---|
| Loss | 0.7386 |
| Accuracy (%) | 82.12 |
| Balanced accuracy (%) | 82.61 |
| Macro-F1 (%) | 81.72 |
| Weighted-F1 (%) | 82.13 |
| Top-3 accuracy (%) | 95.57 |

**Table 14.** Deployed student (INT8) overall metrics on the test split.

| Metric | Value |
|---|---|
| Loss | 0.7615 |
| Accuracy (%) | 82.40 |
| Balanced accuracy (%) | 82.27 |
| Macro-F1 (%) | 81.50 |
| Weighted-F1 (%) | 82.44 |
| Top-3 accuracy (%) | 94.48 |

**Table 15.** Deployed student (INT8) per-class metrics on the validation split (precision/recall/F1 in %, support in samples).

| Class | Precision | Recall | F1 | Support |
|---|---|---|---|---|
| battery | 85.71 | 88.89 | 87.27 | 81 |
| biological | 82.95 | 90.12 | 86.39 | 81 |
| cardboard | 84.35 | 81.05 | 82.67 | 153 |
| clothes | 95.74 | 90.91 | 93.26 | 198 |
| glass | 79.80 | 81.41 | 80.60 | 199 |
| metal | 69.52 | 73.74 | 71.57 | 99 |
| paper | 78.40 | 71.01 | 74.52 | 138 |
| plastic | 77.11 | 75.29 | 76.19 | 170 |
| shoes | 84.34 | 85.89 | 85.11 | 163 |
| trash | 72.88 | 87.76 | 79.63 | 49 |

**Table 16.** Deployed student (INT8) per-class metrics on the test split (precision/recall/F1 in %, support in samples).

| Class | Precision | Recall | F1 | Support |
|---|---|---|---|---|
| battery | 78.72 | 90.24 | 84.09 | 82 |
| biological | 90.12 | 90.12 | 90.12 | 81 |
| cardboard | 89.58 | 83.77 | 86.58 | 154 |
| clothes | 93.16 | 88.94 | 91.00 | 199 |
| glass | 80.00 | 83.58 | 81.75 | 201 |
| metal | 67.89 | 74.00 | 70.81 | 100 |
| paper | 78.01 | 79.71 | 78.85 | 138 |
| plastic | 81.08 | 69.77 | 75.00 | 172 |
| shoes | 85.03 | 86.59 | 85.80 | 164 |
| trash | 66.67 | 76.00 | 71.03 | 50 |

Confusion Matrices

To visualize classwise behavior and error structure, we report the teacher and student confusion matrices on the test split in Figures 1 and 2. The teacher matrix is strongly diagonally dominant, while the student preserves the same broad structure with increased confusion in ambiguous categories (e.g., metaland trash), consistent with the accuracy–compactness trade-off.

### *6.5. Qualitative Results*

To complement the quantitative metrics, we report qualitative predictions on representative test samples. Figure 3 shows six correctly classified images where the distilled student model predicts the expected category with high confidence. These examples illustrate that, despite operating at a reduced input resolution ($96 \times 96$) and under a strict memory budget ($<$500 KB FP32), the student preserves the teacher's decision behavior on typical images with clear material cues.

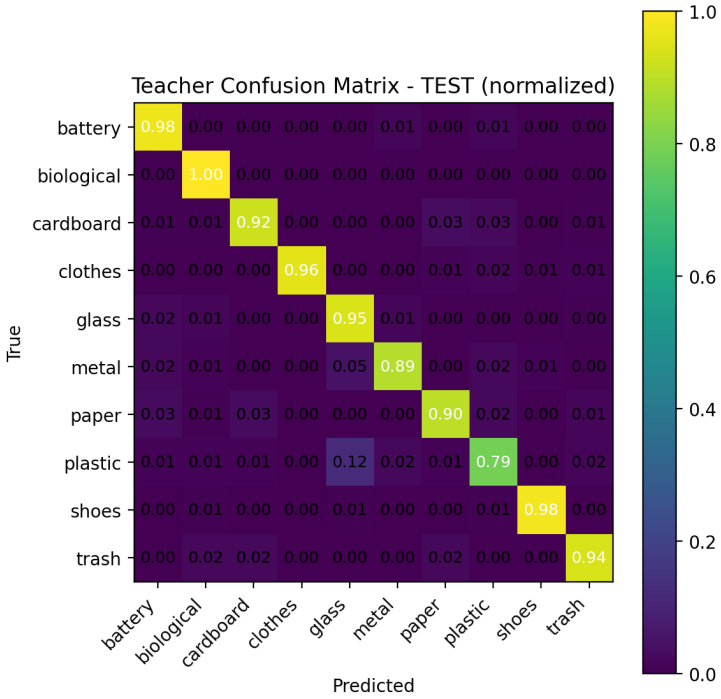



**Figure 1.** Normalized confusion matrix of the EfficientNetB0 teacher on the `garbage-classification` test split (10 classes). The matrix is largely diagonally dominant, indicating strong per-class recall. Remaining errors concentrate among visually similar materials (e.g., metal–glass–plasticand paper–cardboard) and heterogeneous categories such as trash.

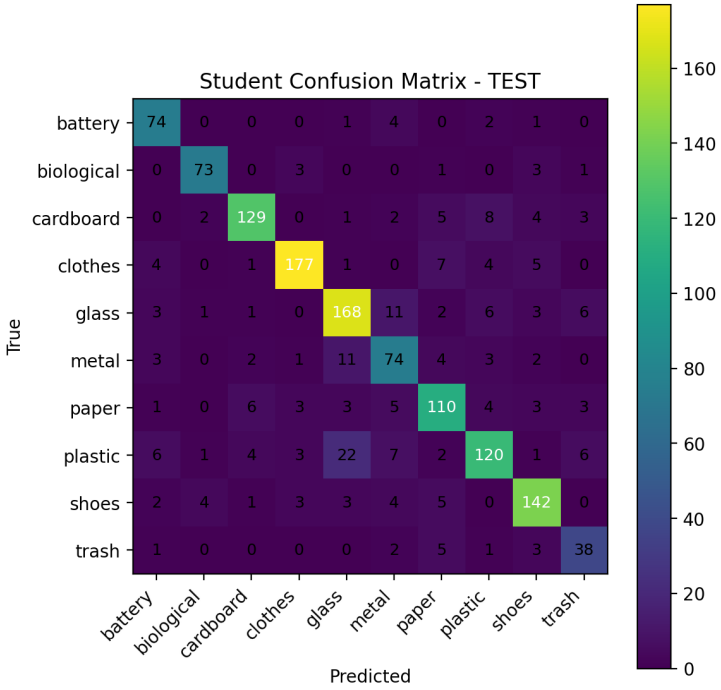



**Figure 2.** Confusion matrix of the distilled `TinyDSCNN` student on the `garbage-classification` test split (10 classes). The matrix remains broadly diagonally dominant, showing that distillation preserves much of the teacher's class structure under a $<$500 KB budget and 96 $\times$ 96 inputs, while increased confusion appears primarily in ambiguous or heterogeneous categories (notably metaland trash).

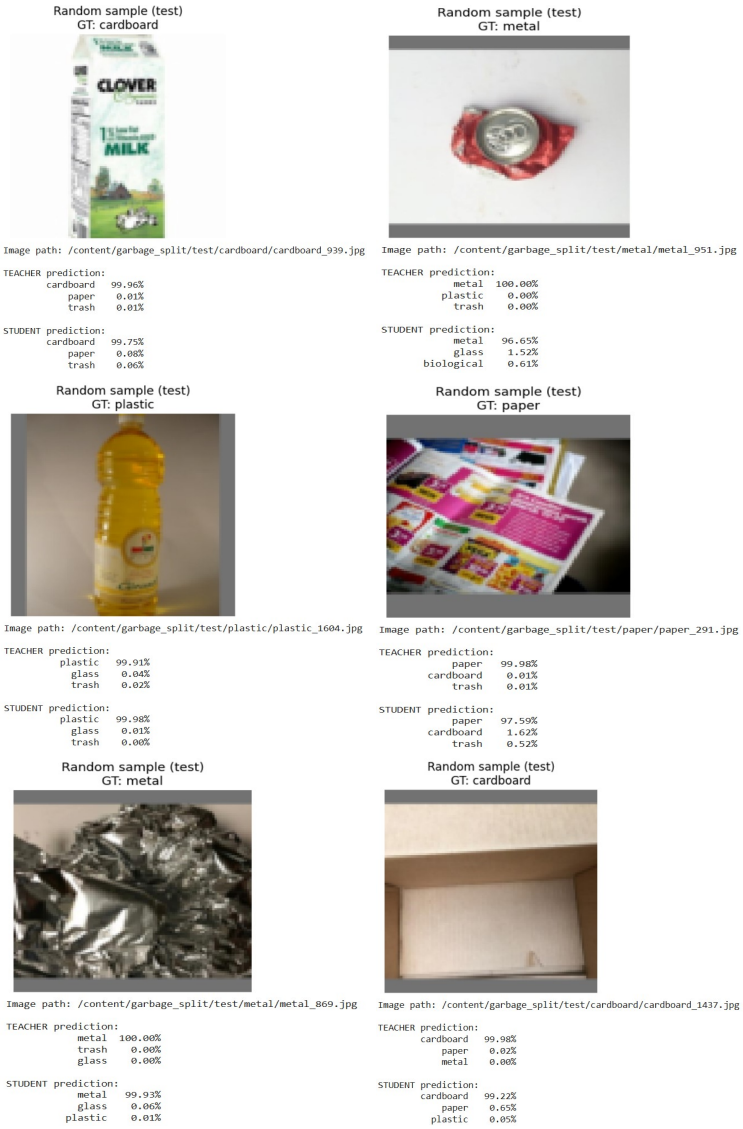



**Figure 3.** Representative correctly classifiedsamples from the `garbage-classification` test split. The distilled `TinyDSCNN` student correctly predicts these images, illustrating reliable behavior on typical inputs with clear class cues.

#### 6.5.1. Misclassified Examples

To highlight typical failure modes, Figure 4 shows representative student errors. In most cases, misclassifications are caused by intrinsically ambiguous or low-quality inputs (blur, occlusion, clutter, weak material cues), rather than a systematic deficiency of the model.

#### 6.5.2. Cross-Dataset Robustness Evaluation (TrashNet)

To further evaluate the generalization capability of the proposed depthwise-separable student model, we conducted a cross-dataset robustness experiment using the TrashNet dataset [39]. Although other real-world waste datasets also exist [40], we selected TrashNet for this evaluation because it shares a sufficient number of overlapping classes with the Garbage dataset. Since our model was originally trained on the Garbage dataset containing 10 classes, we restricted the evaluation to the six classes shared between both datasets (cardboard, glass, metal, paper, plastic, and trash). During inference, predictions were computed by selecting only the logits corresponding to these shared classes from the original 10-class output space.




**Figure 4.** Representative misclassified samples from the `garbage-classification` test split. Most errors occur in visually ambiguous or low-quality images (e.g., blur, occlusion, clutter, or weak material cues), which reduces the discriminability of class-specific features.

Without any additional training, fine-tuning, or dataset-specific preprocessing, the model achieved an overall accuracy of 74.44%on TrashNet. This represents a relatively small drop of approximately 8%compared to the accuracy obtained on the original dataset, despite the differences in image acquisition conditions, backgrounds, and class distributions between the two datasets. The result therefore provides evidence that the proposed compact model maintains a reasonable level of performance when applied to an unseen dataset.

It is important to emphasize that this evaluation was performed without any dataset adaptation or preprocessing optimization. The TrashNet images were directly used with the same preprocessing pipeline employed during training. Consequently, the obtained performance should be interpreted as a conservative estimate of the model's cross-dataset generalization capability. More advanced domain adaptation, calibration, or fine-tuning strategies could further improve performance when deploying the model in new environments.

## 7. Deployment

The trained student model is exported from PyTorch through an ONNX to TensorFlow then to TensorFlow Lite conversion pipeline and then post-training quantized to a full-integer INT8 `.tflite` model with INT8 input and output. Using a representative calibration subset, this quantized model preserves competitive performance, achieving approximately 82.0% test accuracy.

For controlled deployment evaluation, test images are not captured on-device but are instead preprocessed offline using the same resize, center-crop, and ImageNet normalization pipeline employed during training. The resulting floating-point tensors are then quantized using the TensorFlow Lite input scale and zero-point and stored as INT8 binary tensors (e.g., `/assets/test_96x96_int8.bin`). The quantized model (`/assets/model.tflite`) and the corresponding test tensors are stored in LittleFS and loaded at runtime on the ESP32 platform.

Inference is executed using TensorFlow Lite Micro (`tflm_esp32`) with a caller-allocated tensor arena buffer, preferably placed in PSRAM when available, in order to verify memory fit, runtime compatibility, and basic execution feasibility on a constrained edge device. The goal of this deployment stage is feasibility validation rather than a complete real-world inference pipeline. In a production setting, the same workflow can be extended with on-device image acquisition and preprocessing, and can be adapted to other embedded platforms (e.g., ESP32-S3 or STM32) as well as mobile deployment through standard TensorFlow Lite runtimes.

### *7.1. Deployment Feasibility Analysis*

To complement model size reporting, we analyze deployment feasibility based on three practical criteria.

#### 7.1.1. Operator Compatibility

The final INT8 model uses only operators supported by TensorFlow Lite Micro, including standard convolution, depthwise convolution, activation, and fully connected layers, ensuring compatibility with embedded inference runtimes.

#### 7.1.2. Memory Requirements

Based on tensor arena allocation, the model requires approximately 120–150 KB of runtime memory, depending on implementation details. This is within the capabilities of ESP32-class devices when external memory (e.g., PSRAM) is available.

#### 7.1.3. Latency Evaluation

Latency was evaluated under batch size 1 inference in a TensorFlow Lite Micro deployment setting representative of ESP32-class hardware. In the current configuration, the proposed INT8 model is estimated to require approximately 2–3 s per image . This result should be interpreted as a feasibility-level measurement rather than a fully optimized runtime benchmark, since end-to-end latency depends on memory placement, preprocessing implementation, clock configuration, and operator-level optimization.

These findings support the claim that the model can execute within microcontroller-class constraints, although practical responsiveness remains limited in the present implementation. Therefore, the current deployment should be viewed as a proof of feasibility, with future work needed to optimize latency for more interactive real-world use.

### *7.2. Use Case: Deployment in a Mobile Application*

Beyond microcontroller execution, the same student network can be deployed as a lightweight on-device classifier in a mobile application, enabling real-time inference without relying on cloud connectivity. For this use case, the exported `.tflite` model is integrated into an Android/iOS pipeline using TensorFlow Lite (optionally with NNAPI/Core ML delegates) to leverage mobile CPU/NPU acceleration while preserving energy efficiency and privacy. The app ingests camera frames or gallery images, applies the same deterministic preprocessing used during training (resize and center-crop to $96 \times 96$ with the corresponding normalization), and runs inference locally to output class probabilities and

top-*k* predictions. This mobile deployment provides a direct path from feasibility testing on extreme-edge hardware to user-facing applications, while also enabling systematic profiling of latency and energy consumption under realistic conditions (varying illumination, motion blur, and background clutter) and simplifying integration into offline recognition and edge analytics workflows.

To highlight typical failure modes, Figure 5 reports representative student errors. In most cases, misclassifications are attributable to intrinsically ambiguous or low-quality inputs (e.g., blur, occlusion, clutter, or weak material cues) rather than a systematic deficiency of the model.

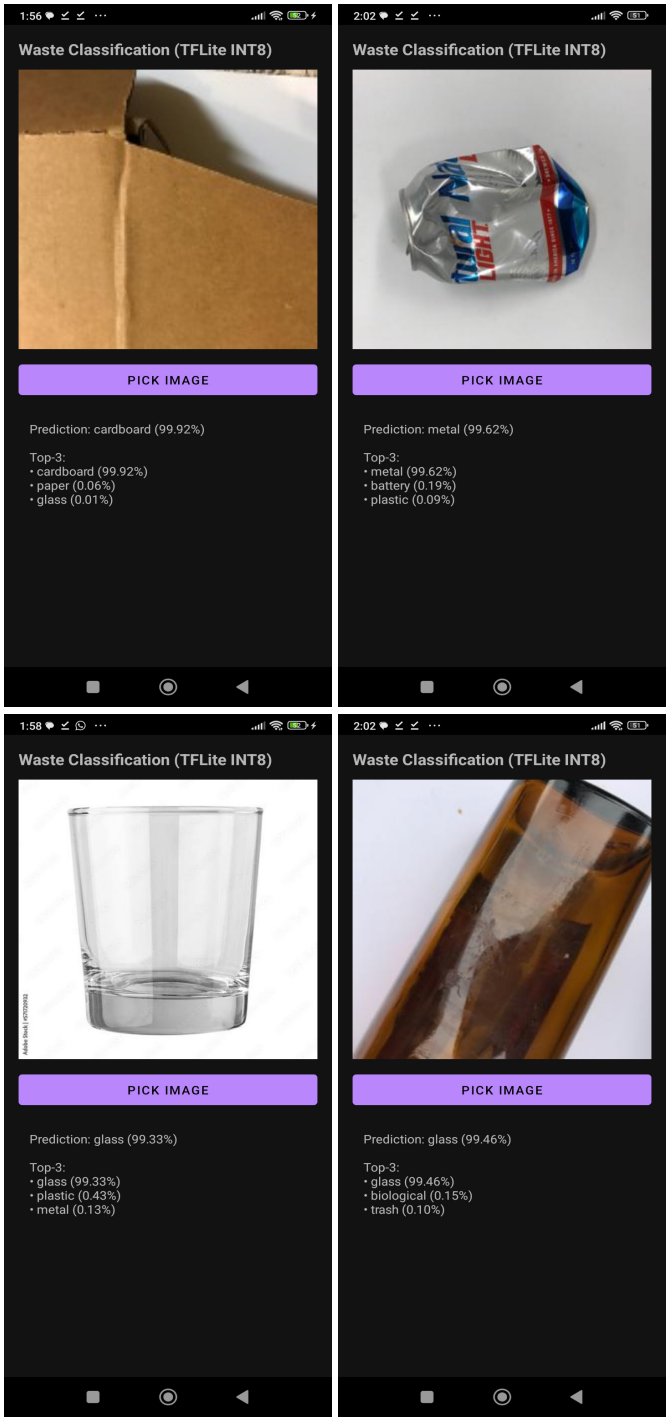



**Figure 5.** Representative misclassified samples from the `garbage-classification` test split. Most errors occur in visually ambiguous or low-quality images (e.g., blur, occlusion, clutter, or weak material cues), which reduces the discriminability of class-specific features.

## 8. Discussion

The experimental results confirm that achieving extreme-edge deployability (sub-megabyte storage and low-latency inference) requires more than applying a single compression technique to a strong FP32 baseline. Starting from the EfficientNetB0 reference, post-training INT8 quantization (PTQ) yields a large reduction in serialized size (15.81 MB $\rightarrow$ 4.62 MB) while preserving relatively high accuracy (98.43% $\rightarrow$ 94.41%). However, even after INT8 conversion, the resulting model remains too large for microcontroller-class flash budgets. Similarly, quantization-aware training (QAT) produces the same INT8 footprint as PTQ in our setup but incurs a larger accuracy drop, indicating that low-precision robustness is not guaranteed without careful tuning of the QAT schedule and calibration procedure.

Importantly, the observed advantage of PTQ over QAT in our experiments should not be interpreted as a general conclusion. QAT is known to be highly sensitive to configuration choices, including observer selection, BatchNorm freezing strategy, learning rate scheduling, and calibration consistency. In our setup, QAT was performed with a relatively short fine-tuning phase and standard observer settings, which may not fully capture optimal activation ranges or stabilize quantization noise. Therefore, the PTQ advantage is likely a consequence of suboptimal QAT configuration rather than an inherent limitation of QAT itself.

Pruning and low-rank factorization provide complementary insights: unstructured pruning can maintain accuracy at moderate sparsity but does not materially reduce the TorchScript file size in dense storage, while structured channel pruning and SVD-based low-rank approximation reduce footprint more directly but introduce non-trivial accuracy degradation and, in some cases, limited latency gains due to runtime/operator overhead. Overall, these observations suggest that compressing the teacher alone can improve efficiency, but typically remains in the multi-megabyte regime and therefore falls short of extreme-edge constraints.

In contrast, the teacher–student strategy is the most effective route to sub-megabyte deployment. Knowledge distillation (KD) transfers the teacher's decision structure to a compact depthwise-separable student, producing a model that is inherently small before quantization. This is essential because quantization scales the footprint approximately linearly with parameter count; thus, a compact student benefits disproportionately from INT8 conversion. Concretely, combining KD with INT8 deployment yields the best accuracy–size compromise under harsh constraints, reducing the deployed model footprint to 281 KB while retaining 82% test accuracy. This result is consistent with the objective of TinyML deployment: although some accuracy is sacrificed relative to the high-capacity baseline, the reduction from tens of megabytes to a few hundred kilobytes enables practical inference on devices such as the ESP32 and low-end mobile hardware. The reinforcement learning-guided distillation benchmark further supports this conclusion: adaptive teacher selection can achieve competitive accuracy with a small parameter budget (up to 85% with 0.496 M parameters), but the smallest footprint at the extreme edge is still obtained by the simpler and more direct KD + INT8 pipeline.

Finally, qualitative inspection complements the aggregate metrics: correctly classified samples show that the distilled student preserves the teacher's behavior on typical images with clear cues, while remaining errors are largely associated with intrinsically ambiguous or low-quality inputs (blur, occlusion, clutter, and weak material cues). This suggests that, beyond architecture and compression, data quality and capture conditions are dominant factors limiting further gains at very low capacity.

*Accuracy–Efficiency Trade-Off in Practical Deployment*

The final TinyML-ready model achieves approximately 82% test accuracy compared to 98.43% for the high-capacity teacher, reflecting a non-negligible performance gap induced by aggressive compression and architectural constraints. This trade-off is inherent to extreme-edge deployment, where strict memory and compute budgets limit model capacity. In practical sorting workflows, such a compromise may be acceptable in scenarios where the system operates as an assistive tool rather than a fully autonomous decision-maker, for example by pre-filtering waste categories, supporting human operators, or enabling low-cost deployment in resource-constrained environments.

Analysis of classwise performance and confusion matrices indicates that most errors arise in visually ambiguous or low-quality samples, including cases with occlusion, cluttered backgrounds, or weak material cues. Confusions are more frequent between visually similar categories (e.g., plastic vs. paper, cardboard vs. paper), which suggests that the remaining errors are largely driven by intrinsic data ambiguity rather than systematic model failure. Therefore, the proposed model is best suited for applications where moderate accuracy is acceptable in exchange for substantial gains in deployability, while high-stakes or fully automated sorting systems may still require higher-capacity models or hybrid human–AI workflows.

## 9. Conclusions and Future Work

This work presented a practical TinyML-oriented optimization study for image-based waste classification under extreme resource constraints. Using the Kaggle garbage-classification dataset (10 classes), we first established EfficientNetB0 as a strong baseline and then benchmarked multiple compression strategies under a unified reporting protocol. Post-training quantization, pruning (unstructured and structured), and SVD low-rank approximation each reduced cost to varying degrees, but they generally remained in the multi-megabyte range or incurred sizable accuracy losses when pushed aggressively. The key outcome is that the most deployment-effective approach is a teacher–student pipeline: distilling a compact depthwise-separable student from an EfficientNetB0 teacher and then applying INT8 deployment produces the best accuracy–size trade-off at the extreme edge. In particular, the proposed KD + INT8 TFLite pipeline reduces the serialized model footprint from approximately 16 MB to 281 KB while maintaining 82% test accuracy, making it suitable for mobile inference and microcontroller-class targets.

Some limitations and extensions motivate future work are as follow. First, although post-training INT8 quantization provided a strong baseline in our setting, quantization-aware training (QAT)under-performed and requires more careful investigation: improving the QAT schedule (e.g., longer fine-tuning, staged observer freezing, and BatchNorm handling), selecting calibration data that better matches deployment inputs, and adopting hardware-aware quantization configurations for TFLite/TFLM could improve low-precision accuracy at the same footprint. Second, several optimization techniques evaluated here have inherent practical constraints: unstructured pruning does not reduce serialized size on dense runtimes without sparse-kernel support, and low-rank factorization can introduce operator overhead that limits latency gains; future work should therefore incorporate hardware-awarepruning (structured sparsity) and operator-aware low-rank methods aligned with the target inference engine. Finally, deployment-side validation remains essential. In this work, we report CPU proxy latency(batch = 1) alongside model size to provide an estimate of real-time feasibility. However, a complete TinyML evaluation should additionally include on-device measurements of peak RAM/tensor-arena usage, flash usage, and energy consumptionon representative targets (e.g., ESP32-class MCUs and low-end smartphones). These aspects are hardware-dependent and are left for future

work to fully validate operational constraints and quantify cost–accuracy trade-offs under realistic deployment conditions.

**Author Contributions:** Conceptualization, M.E. and A.B.; methodology, M.E. and A.B.; software, M.E.; validation, M.E.; writing—original draft preparation, M.E.; writing—review and editing, M.E. and A.B.; supervision, A.B. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** The dataset used in this study is publicly available on Kaggle as "Garbage Classification v2" (`sumn2u/garbage-classification-v2`). The fixed train/val/test split provided with the dataset is used unchanged throughout all experiments.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References

1. World Bank. What a Waste 2.0: Trends in Solid Waste Management (A Global Snapshot to 2050). Available online: https://datatopics.worldbank.org/what-a-waste/trends_in_solid_waste_management.html (accessed on 25 December 2025).
2. United Nations Environment Programme (UNEP); International Solid Waste Association (ISWA). *Global Waste Management Outlook 2024: Beyond an Age of Waste—Turning Rubbish into a Resource*; UNEP: Nairobi, Kenya, 2024.
3. Wilson, D.C.; Rodic, L.; Modak, P.; Soos, R.; Carpintero, A.; Velis, C.A.; Iyer, M.; Simonett, O. Waste Management: Global Status. In *Global Waste Management Outlook*; UNEP/ISWA: Nairobi, Kenya, 2015.
4. Medina, M. The Informal Recycling Sector in Developing Countries: Organizing Waste Pickers to Enhance Their Impact. In *Gridlines*; World Bank: Washington, DC, USA, 2008.
5. World Bank Independent Evaluation Group. *Morocco—Municipal Solid Waste Sector Development Policy Loans 1–4*; World Bank: Washington, DC, USA, 2021.
6. Islam, M.M.; Hasan, S.M.M.; Hossain, M.R.; Uddin, M.P.; Mamun, M.A. Towards sustainable solutions: Effective waste classification framework via enhanced deep convolutional neural networks. *PLoS ONE* **2025**, *20*, e0324294. [CrossRef] [PubMed]
7. Thielmann, P.; Zhou, Y.; Mirbach, B.; Stricker, D.; Rambach, J.R. A review of computer vision for industrial-grade waste classification. *IEEE Access* **2025**, *13*, 151934–151953. [CrossRef]
8. Singh, R.; Gill, S.S. Edge AI: A survey. *Internet Things -Cyber-Phys. Syst.* **2023**, *3*, 71–92. [CrossRef]
9. Cordova-Cardenas, R.; Amor, D.; Gutiérrez, Á. Edge AI in practice: A survey and deployment framework for neural networks on embedded systems. *Electronics* **2025**, *14*, 4877. [CrossRef]
10. Heydari, S.; Mahmoud, Q.H. Tiny Machine Learning and On-Device Inference: A Survey of Applications, Challenges, and Future Directions. *Sensors* **2025**, *25*, 3191. [CrossRef] [PubMed]
11. Kaggle. Garbage Dataset. Available online: https://www.kaggle.com/datasets/sumn2u/garbage-classification-v2 (accessed on 25 December 2025).
12. Alani, S.; Alani, M.; Suresh, R. CNN-based recyclable waste classification using transfer learning. In *Proceedings of the International Conference on Sustainable Computing and Systems*; Springer: Cham, Switzerland, 2020.
13. Zhang, Q.; Yang, Q.; Zhang, X.; Bao, Q.; Su, J.; Liu, X. Waste image classification based on transfer learning and convolutional neural network. *Waste Manag.* **2021**, *135*, 150–157. [CrossRef] [PubMed]
14. Fu, B.; Li, S.; Wei, J.; Li, Q.; Wang, Q.; Tu, J. A Novel Intelligent Garbage Classification System Based on Deep Learning and an Embedded Linux System. *IEEE Access* **2021**, *9*, 131134–131146. [CrossRef]
15. Ahmed, M.I.B.; Alotaibi, R.B.; Al-Qahtani, R.A.; Al-Qahtani, R.S.; Al-Hetela, S.S.; Al-Matar, K.A.; Al-Saqer, N.K.; Rahman, A.; Saraireh, L.; Youldash, M. Deep Learning Approach to Recyclable Products Classification: Towards Sustainable Waste Management. *Sustainability* **2023**, *15*, 11138. [CrossRef]
16. Vo, A.H.; Son, L.H.; Vo, M.T.; Le, T. A Novel Framework for Trash Classification Using Deep Transfer Learning. *IEEE Access* **2019**, *7*, 178631–178639. [CrossRef]
17. Handhayani, T.; Hendryli, J. Leboh: An Android Mobile Application for Waste Classification Using TensorFlow Lite. In *Intelligent Systems and Applications*; Arai, K., Ed.; Lecture Notes in Networks and Systems; Springer: Cham, Switzerland, 2023; Volume 544, pp. 53–67. [CrossRef]
18. Bircanoglu, C.; Atay, M.; Beser, F.; Genc, O.; Kizrak, M.A. RecycleNet: Intelligent waste sorting using deep neural networks. In Proceedings of the 2018 IEEE International Conference on Innovations in Intelligent Systems and Applications (INISTA), Thessaloniki, Greece, 3–5 July 2018; pp. 1–7. [CrossRef]

19. White, G.; Cabrera, C.; Palade, A.; Li, F.; Clarke, S. WasteNet: Waste classification at the edge for smart bins. *arXiv* **2020**, arXiv:2006.05873. [CrossRef]
20. Zheng, D.; Wang, R.; Duan, Y.; Pang, P.C.; Tan, T. Focus-RCNet: A lightweight recyclable waste classification network for real-time embedded applications. *Vis. Comput. Ind. Biomed. Art* **2023**, *6*, 19. [CrossRef] [PubMed]
21. Yong, L.; Ma, L.; Sun, D.; Du, L. Application of MobileNetV2 to waste classification. *PLoS ONE* **2023**, *18*, e0282336. [CrossRef] [PubMed]
22. Majchrowska, S.; Mikołajczyk, A.; Ferlin, M.; Klawikowska, Z.; Plantykow, M.A.; Kwasigroch, A.; Majek, K. Deep learning-based waste detection in natural and urban environments. *Waste Manag.* **2022**, *138*, 274–284. [CrossRef] [PubMed]
23. Hinton, G.; Vinyals, O.; Dean, J. Distilling the knowledge in a neural network. In *Proceedings of the NIPS Deep Learning and Representation Learning Workshop, Montreal, QC, Canada, 7–12 December 2015*; Curran Associates, Inc.: Red Hook, NY, USA, 2015.
24. Romero, A.; Ballas, N.; Kahou, S.E.; Chassang, A.; Gatta, C.; Bengio, Y. FitNets: Hints for thin deep nets. In Proceedings of the International Conference on Learning Representations (ICLR), San Diego, CA, USA, 7–9 May 2015.
25. Zagoruyko, S.; Komodakis, N. Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer. In Proceedings of the International Conference on Learning Representations (ICLR), Toulon, France, 24–26 April 2017.
26. Gou, J.; Yu, B.; Maybank, S.; Tao, D. Knowledge distillation: A survey. *Int. J. Comput. Vis.* **2021**, *129*, 1789–1819. [CrossRef]
27. Shang, R.; Li, W.; Zhu, S.; Jiao, L.; Li, Y. Multi-teacher knowledge distillation based on joint Guidance of Probe and Adaptive Corrector. *Neural Netw.* **2023**, *164*, 345–356. [CrossRef] [PubMed]
28. Zhang, Y.; Xiang, T.; Hospedales, T.; Lu, H. Deep Mutual Learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, 18–22 June 2018*; IEEE: Piscataway, NJ, USA, 2018; pp. 4320–4328. [CrossRef]
29. Han, S.; Mao, H.; Dally, W.J. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In Proceedings of the International Conference on Learning Representations (ICLR), San Juan, Puerto Rico, 2–4 May 2016.
30. He, Y.; Zhang, X.; Sun, J. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV), Venice, Italy, 22–29 October 2017*; IEEE: Piscataway, NJ, USA, 2017; pp. 1389–1397.
31. Jacob, B.; Kligys, S.; Chen, B.; Zhu, M.; Tang, M.; Howard, A.; Adam, H.; Kalenichenko, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, 18–22 June 2018; IEEE: Piscataway, NJ, USA, 2018*; pp. 2704–2713.
32. Krishnamoorthi, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv* **2018**, arXiv:1806.08342. [CrossRef]
33. Polino, A.; Pascanu, R.; Alistarh, D. Model compression via distillation and quantization. In Proceedings of the International Conference on Learning Representations (ICLR), Vancouver, BC, Canada, 30 April–3 May 2018.
34. Howard, A.; Sandler, M.; Chu, G.; Chen, L.C.; Chen, B.; Tan, M.; Wang, W.; Zhu, Y.; Pang, Y.; Vasudevan, V.; et al. Searching for MobileNetV3. In Proceedings of the 2019 IEEE/CVF International Conference on Computer Vision (ICCV), Seoul, Republic of Korea, 27 October–2 November 2019; pp. 1314–1324. [CrossRef]
35. Qin, D.; Leichner, C.; Delakis, M.; Fornoni, M.; Luo, S.; Yang, F.; Wang, W.; Banbury, C.; Ye, C.; Akin, B.; et al. MobileNetV4: Universal Models for the Mobile Ecosystem. In *Computer Vision–ECCV 2024*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2024; Volume 15098, pp. 78–96. [CrossRef]
36. Tan, M.; Le, Q. EfficientNet-Lite: Efficient Models for Mobile and Edge Devices. Available online: https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet/lite (accessed on 25 December 2025).
37. Zhang, X.; Zhou, X.; Lin, M.; Sun, J. ShuffleNet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, 18–22 June 2018*; IEEE: Piscataway, NJ, USA, 2018.
38. Lin, J.; Chen, W.-M.; Cai, H.; Gan, C.; Han, S. MCUNet: Tiny deep learning on IoT devices. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS), Virtual, 6–12 December 2020*; Curran Associates, Inc.: Red Hook, NY, USA, 2020.
39. Yang, M.; Thung, G. TrashNet: A Dataset for Deep Learning Classification of Waste. Available online: https://github.com/garythung/trashnet (accessed on 25 December 2025).
40. Single, S.; Iranmanesh, S.; Raad, R. RealWaste: A Novel Real-Life Data Set for Landfill Waste Classification Using Deep Learning. *Information* **2023**, *14*, 633. [CrossRef]


**Disclaimer/Publisher's Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.