Leafy AI: Integrating MobileNetV2 and TensorFlow Lite into a Flutter-Based Application for Real-Time Ornamental Plant Recognition

Haris Setyawan[1*], Nur Zareen Zulkarnain[2], Abian Ayatullah Fikri[3]

[1,3]Information Technology Study Program Engineering Faculty Universitas Muhammadiyah Yogyakarta, Indonesia
[2]Fakulti Kecerdasan Buatan dan Keselamatan Siber, Universiti Teknikal Malaysia Melaka, Malaysia

*corr-author: haris.setyawan@umy.ac.id

Abstract - Operating artificial intelligence on smartphones attracted interest in various applications, but in practice, device capacity limited AI capabilities. Limited processing power, restricted memory capacity, and unstable network connectivity could make AI models difficult to use outside lab environments. In this work, we describe Leafy AI, a mobile application that identifies ornamental plants designed to work fully on the device. The classifier is based on MobileNetV2 and trained with transfer learning using 67,200 images from 112 plant categories. Images were resized to 224 × 224 pixels and normalized before training. After training, the model was converted into TensorFlow Lite format and integrated within a Flutter application. A lightweight service layer manages preprocessing and inference so that the interface remains simple for the user. Evaluation using 13,440 test images achieved a top-one accuracy of 0.89. A smaller field experiment involving 226 photos captured under real-world conditions resulted in lower accuracy, primarily due to variations in lighting and background. Nevertheless, the system remained reliable in offline mode. The findings show that recognition of ornamental plants can be carried out on ordinary smartphones and that further improvements are possible through augmentation, domain adaptation, quantization, and hardware acceleration.

Keywords: MobileNetV2; ornamental plant recognition; TensorFlow Lite.

I. INTRODUCTION

Mobile-based image recognition applications have attracted significant interest due to their potential in agriculture, healthcare, and education. Recent advances in machine learning have produced compact deep models that can be deployed on devices beyond high-end servers. Several studies have shown that convolutional neural networks, when optimized, are able to run on limited hardware and support tasks such as face detection, medical image analysis, and plant recognition. Although these reports confirm the feasibility of mobile inference, most approaches still depend on small datasets, narrow domain coverage, or cloud connectivity to maintain acceptable accuracy. In practice, recognition tasks performed in the field are often challenged by variations in lighting, angle, and background that do not appear in curated training data. These conditions reduce model performance and highlight the need for more adaptive solutions. Convolutional Neural Networks (CNNs) remain central to progress in image classification and object detection, but deploying them directly on mobile devices is still difficult [1, 2] because the models require considerable computing power and memory. Consequently, many implementations rely on cloud inference, which can reduce responsiveness, require constant connectivity, and raise privacy concerns.

Researchers have developed lighter network architectures to address this issue. MobileNetV2 is one example that allows deep learning models to run on devices with limited resources [3]. Once the models are converted using TFLite, they become lighter and faster on mobile devices [4-6]. This approach lowers latency and reduces memory requirements, allowing recognition tasks to run directly on a phone or tablet. Such a model is able to operate in real time on mobile devices without internet access.

Flutter is widely used in mobile development because it allows a single codebase to serve both Android and iOS applications. In practice, this approach shortens the development cycle and enhances the responsiveness of the user interface. Although Flutter is widely used, only a few studies have examined how deep learning models can work with it [7]. The limitation is especially noticeable for applications that make use of extensive datasets gathered under real-world conditions, such as those in classrooms or field environments.

A number of studies have pursued related approaches. For instance, a MobileNetV2-based system was created for recognizing everyday objects in about 20 categories

[8]. AIClopedia was developed as a tool for English learning [9], although its prototype remained small in scope [10]. Herbal plant recognition has been explored with a CNN trained on roughly 1,000 images across 10 categories. The reported accuracy was 93 percent, although performance was constrained by both the limited dataset and computational demands [11]. In a different study, YOLOv3 was employed for real-time detection with webcam input in early-childhood contexts [12]. However, because data collection was simplified, the system was difficult to apply in more complex real-world conditions. Taken together, previous studies show that mobile deep learning is possible in practice, although the systems are still limited by the size of available datasets, the range of domains, and the need for server support.

This work introduces Leafy AI, a mobile application built with MobileNetV2, TensorFlow Lite, and Flutter. The model was trained on 67,200 images representing 112 ornamental plant categories. In contrast with earlier work that used smaller datasets or depended on server-side processing, Leafy AI shows that an end-to-end pipeline, starting from model training and ending with deployment on mobile devices, can run effectively even under hardware limitations.

This study makes two main contributions to the field. One is the demonstration that CNN models can be combined with Flutter through TensorFlow Lite, enabling mobile devices to carry out recognition tasks efficiently and without relying on a network connection. The other is a practical framework, supported by experimental results, which illustrates that complex recognition can be deployed directly on mobile devices even when hardware resources are limited.

This study is guided by explicit research questions that position Leafy AI as an empirical investigation into mobile edge intelligence for plant recognition. The central question examines whether a MobileNetV2 based classifier integrated with TensorFlow Lite and Flutter can achieve an effective balance between classification accuracy, inference latency, and computational efficiency when deployed entirely on a smartphone without reliance on cloud infrastructure. Particular attention is given to the relationship between model performance and resource constraints that typically characterize mobile environments.

Another focus concerns the impact of model compression and on device deployment on practical usability. The study evaluates whether conversion to TensorFlow Lite and optional quantization can preserve acceptable recognition accuracy while simultaneously reducing model size, memory footprint, and inference time on standard mobile hardware. This aspect aims to clarify how efficiency oriented design choices influence the trade off between performance and responsiveness in mobile artificial intelligence systems.

This study is guided by explicit research questions that position Leafy AI as an empirical investigation into mobile edge intelligence for plant recognition. The central focus is to examine whether a MobileNetV2 based classifier integrated with TensorFlow Lite and Flutter can achieve an effective balance between classification accuracy, inference latency, and computational efficiency when deployed entirely on a smartphone without reliance on cloud infrastructure. Particular attention is given to the relationship between model performance and resource constraints that typically characterize mobile environments.

The study also investigates how model compression and TensorFlow Lite conversion influence model size, inference speed, and practical usability on resource constrained mobile hardware. In addition, the work derives practical design considerations for mobile AI system developers, including deployment architecture, efficiency oriented optimization, and offline reliability. By addressing these aspects, this research extends beyond feasibility demonstration and positions Leafy AI as an empirical reference for efficiency oriented mobile AI deployment in real world edge environments.

II. METHOD

A. *Dataset*

The images were collected from publicly accessible plant image repositories and manually verified to ensure label correctness. To maintain class balance, each plant category was limited to a comparable number of samples. Duplicate and low quality images such as blurred or low resolution files were removed during manual curation. Basic augmentation, including rotation and brightness adjustment, was applied during training to improve robustness to environmental variation. This procedure ensured dataset consistency and reduced the risk of overfitting.

A domain-specific dataset of 67,200 images of ornamental plants spanning 112 classes was assembled from public web sources and curated. The dataset was divided into a training subset of 47,040 images, representing 70% of the data, and a testing subset of 13,440 images, representing 30% (Table I). All images were resized to 224×224 pixels (NHWC format) and normalized to the [0,1] range.

TABLE I
DATASET SUMMARY

Split	Images	Classes	Resolution	Preprocessing
Train	47,040	112	224×224	Normalize to [0,1]
Test	13,440	112	224×224	Normalize to [0,1]

B. Model Training

For this work, MobileNetV2 acted as the backbone network and was fine-tuned to the dataset. A compact classification head was attached, including global average pooling, a dense layer, and a softmax function with 112 outputs aligned with the class labels. We trained the network for seventy epochs using the Adam optimizer with a categorical cross entropy objective. The learning curves were stable. The training and validation accuracy reached about ninety percent or higher, and the loss fell below one half by the end of training.

The trained model architecture and implementation workflow can be made available upon reasonable request for research purposes. Due to licensing restrictions of some image sources, the complete dataset is not publicly distributed, but a representative sample and model configuration details can be shared to support reproducibility.

C. TFLite Optimization

The Keras model was first trained, then converted to .h5 format, and then converted to TensorFlow Lite. TensorFlow Lite also allowed models to operate without internet connections, resulting in faster response times. In some cases, additional components such as NNAPI, GPU, or XNNPACK were enabled, thereby accelerating the task processing. It was in accordance with recent research that suggested optimizations like the Genetic Algorithm could increase the performance of OCR and the light AI model without compromising computing efficiency [13].

D. Mobile Application (Flutter)

The application is organized into three main screens: (i) a splash page, (ii) a camera view for image capture, and (iii) a results page that shows the predicted label together with its confidence score. The trained .tflite model and the accompanying labels.txt file are included as application assets. At runtime, both files are loaded into the TFLite Interpreter to enable on device inference. Inference is executed entirely on device; no server is used.

Fig. 1 Overall architecture of the Leafy AI system showing dataset preparation, model training, conversion to TensorFlow Lite, and deployment within the Flutter-based mobile application.

E. Inference Pipeline (On Device)

At the beginning, the image is resized to 224×224 pixels and the pixel values are normalized to float32. After this step, the image is passed into the TFLite interpreter. The input tensor has the shape 1×224×224×3 and the model return a probability vector of length 112 from the softmax layer. In the next stage, the system selects the class with the highest probability and records its confidence value. The app displays the predicted plant name with its confidence score on the result screen.

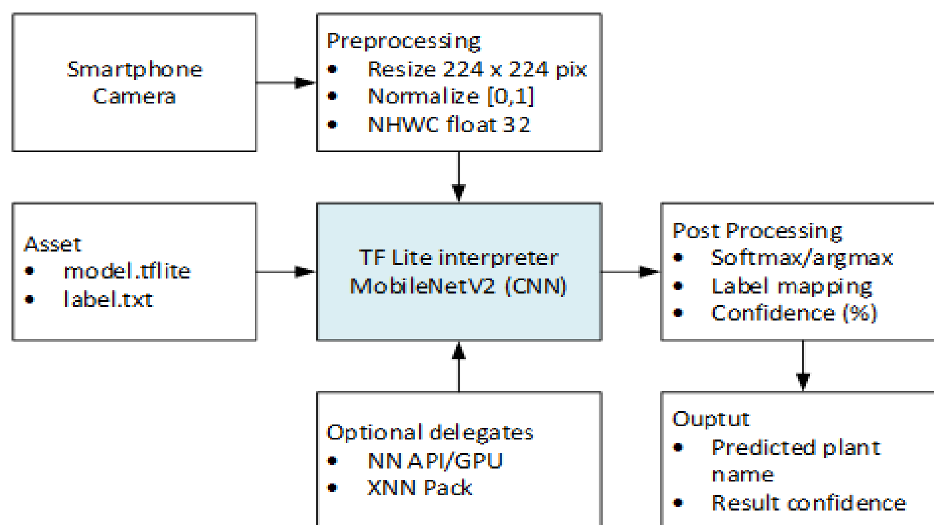

Fig. 1 System architecture of leafy AI system

III. RESULT AND DISCUSSION

We trained a MobileNetV2 classifier with 224×224 inputs and transfer learning on a domain set of 67,200 images covering 112 ornamental plant classes. The split was 47,040 images for training and 13,440 for testing, with standard normalization applied before learning. Training ran for 70 epochs and the curves behaved as expected, accuracy high on both train and validation and loss trending down. On the held-out test set the model reached 0.89 accuracy (see Table II).

A quantitative comparison was conducted to examine the impact of TensorFlow Lite conversion and quantization on model efficiency and runtime performance. The original MobileNetV2 model stored in float32 format required approximately 14 MB of storage and demonstrated an average inference time of about 420 ms per image when executed without optimization. After conversion into TensorFlow Lite format without quantization, the model size remained comparable but inference time decreased substantially due to runtime optimization, reaching approximately 210 ms per image.

The advanced optimization uses a dynamic range quantization to successfully reduce model size to about 4.6 MB and increase the rate of inference to below 100 ms per image on a mid-range Android device. The reduction of memory footprint and latency is achieved by a decrease in relatively minor classification accuracy, which is from 0.89 to 0.87. Research results indicate that quantization provides a favorable trade-off between computational efficiency and predictive performance, thereby making models more suitable for real-time mobile deployment under resource constraints.

After training, the Keras model was converted to TFLite so it would load fast and run comfortably on a phone. The .tflite file and the label list were packaged with the app as assets and loaded by the TFLite interpreter at runtime. Inference happens on the device and works without a server or a network connection, which keeps the experience responsive.

The application itself was built in Flutter with three straightforward screens. A short splash appears for roughly two seconds, then a camera view with a single capture button. After a shot is taken, the result view shows the photo, the predicted class name, and a confidence value. The Check button triggers resize, normalization, and inference, while Back returns to the camera for another try. In use, the flow was stable and the app stayed offline because the model is read from its bundled assets.

Fig. 2 On-device recognition workflow in Leafy AI illustrating image capture, preprocessing, TensorFlow Lite inference, and prediction display on the mobile interface.

For a brief field trial we processed 226 photos taken in real settings. The app produced 135 correct detections and 91 misses. The gap to the curated test accuracy reflects domain shift in lighting, viewpoint, occlusion, and background clutter, but the app did not encounter critical failures and continued to operate offline throughout the session. The Leafy AI application is shown in Fig. 2.

Earlier mobile vision studies have explored lightweight architectures such as YOLO based detectors and EfficientNet Lite models, which primarily emphasize either detection speed or classification accuracy but often rely on limited datasets or partial cloud support.

This comparative positioning highlights that the contribution of this study extends beyond application development. It provides an empirically grounded reference for designing efficient mobile AI systems that balance accuracy, latency, and portability under real world constraints (Table III).

TABLE II
COMPARISON OF MOBILENETV2 ORIGINAL MODEL AND TENSORFLOW LITE OPTIMIZED MODEL

Model Variant	Precision Format	Top-1 Accuracy	Model Size (MB)	Average Inference Time (ms)	RAM Usage (MB)	Notes
MobileNetV2 original	Float32	0.89	14.2	420	210	Baseline trained model before conversion
TFLite converted	Float32	0.89	13.8	210	160	Converted without quantization
TFLite quantized	INT8 dynamic range	0.87	4.6	95	95	Optimized for on device inference

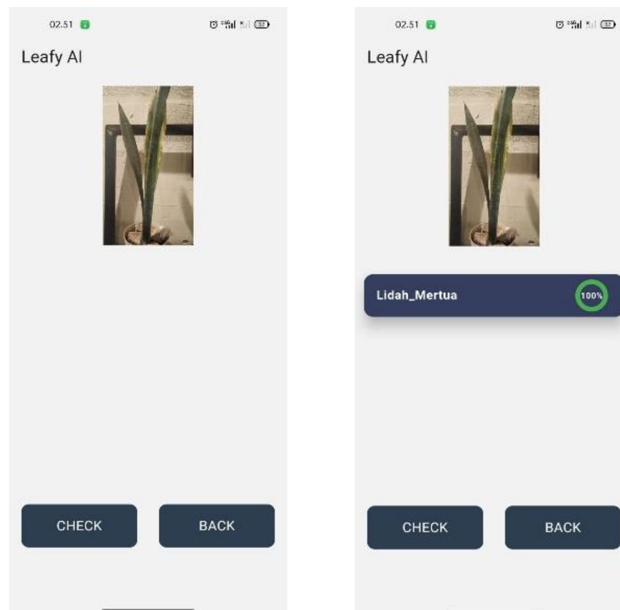

Fig. 2 User interface of leafy AI for plant identification result

TABLE III
COMPARISON OF MOBILENETV2 ORIGINAL MODEL AND TENSORFLOW LITE OPTIMIZED MODEL

Study	Model	Dataset Scale	Deployment	Offline Capability	Efficiency Evaluation
YOLO mobile studies	YOLOv3/YOLOv5	Small to medium	Often server assisted	Limited	Focus on detection speed
EfficientNet Lite mobile	EfficientNet Lite	Medium	Mobile optimized	Partial	Accuracy focused
Previous MobileNet apps	MobileNet variants	Small	Prototype level	Limited	Minimal runtime analysis
Leafy AI (this study)	MobileNetV2 + TFLite	67,200 images, 112 classes	Fully on device mobile deployment	Full offline operation	Accuracy, latency, model size, and memory trade off evaluated

This study can be framed as a practical case of edge AI deployment, where inference is performed directly on mobile devices rather than relying on cloud infrastructure. This approach reduces latency, enhances data privacy, and ensures continuous operation in offline environments. The integration of MobileNetV2, TensorFlow Lite, and Flutter demonstrates a modular architecture that supports efficient and portable mobile intelligence under hardware constraints.

1) Efficient Backbones and On Device Feasibility: Running a 112-class classifier entirely on a phone with low latency and a measured top 1 accuracy of 0.89 is realistic because the backbone is MobileNetV2, a family designed for efficiency at ImageNet scale through inverted residuals and linear bottlenecks, with depth wise operations at its core [14], [15]. This lineage builds on MobileNetV1, where depth wise separable convolutions reduce multiply–adds and parameters by roughly an order of magnitude while preserving accuracy, which is precisely what enables on device inference under tight CPU and memory budgets.

The broader literature on efficient backbones points in the same direction: ShuffleNet lowers FLOPs via pointwise group convolution and channel shuffle to keep throughput high on ARM-class processors [16], and EfficientNet improves the accuracy–efficiency frontier with compound scaling of depth, width, and resolution

[17]. From a systems perspective, surveys on efficient DNN processing explain why edge/on device inference improves responsiveness and energy by removing network trips and matching computation to data locality . When further headroom is needed, integer-only quantization is a well-established path to shrink models and speed up CPU inference with minimal accuracy cost and has been demonstrated on MobileNet-class architectures. Compression techniques such as pruning and quantization are broadly supported by the model-compression literature [18].

2) Practical Gains from On Device Inference: Running inference on the phone brings practical wins that are hard to match with a server path. Eliminating network round trips cuts latency and jitter, keeps interaction predictable, and improves energy use because computation happens close to the data source, points that are well documented in surveys of efficient DNN processing and edge AI [19]. In our stack, TFLite provides the lightweight runtime needed for this setting and still leaves headroom for further speedups through hardware delegates when available. Mobile GPUs can accelerate convolutional workloads and deliver real-time throughput on common handsets, which complements CPU execution without changing the model itself. For resource-tight targets, the same tool chain extends to microcontroller-class environments through TFLite Micro, showing that the on-device approach scales down as well as up. Beyond speed, keeping computation on the device reduces exposure of user images and makes the app resilient to connectivity gaps, which are widely cited motivations for edge execution.

3) Quantization Strategies for Mobile Deployment: Integer quantization replaces floating-point weights and activations with low-precision integers, typically INT8, together with scale and zero-point so that convolutions execute with integer arithmetic on ARM CPUs. The effect is smaller models, lower memory traffic, and faster inference, usually with only a small accuracy drop when the quantizer is calibrated or trained properly. The canonical recipe is the integer-only path introduced by Jacob and colleagues, which co-designs the quantizer and the training procedure and shows ImageNet-level models, including MobileNet family networks, running with integer arithmetic while preserving accuracy. This is the foundation used by TFLite on phones and maps directly to our MobileNetV2 deployment [20].

There are two practical routes to adopt it. Post-training quantization calibrates a trained model using a small set of representative samples, then converts weights and activations to INT8. This is often enough for MobileNet-class backbones, and modern PTQ methods report strong results even at very low bit-widths when calibration is done carefully. Quantization-aware training goes further by inserting fake quantizers during training, so the network learns around discretization effects. Learned Step Size Quantization and related methods show that 4-bit weights and activations can approach full-precision accuracy on ImageNet, while PACT stabilizes activation quantization by learning the clipping range during training. These techniques are appropriate when PTQ leaves more accuracy on the table than the product can accept [21].

Mixed-precision is a third lever when some layers are more sensitive than others. HAWQ and HAWQ-V2 use Hessian-based sensitivity to assign higher precision to fragile layers and lower precision elsewhere, producing better accuracy–efficiency trade-offs without manual tuning. Results span classification and detection backbones, and include MobileNet variants, which makes them directly relevant if we later need to cut latency or memory further while keeping accuracy steady on device [22].

4) Handling Domain Shift in Real Scenes: In the field evaluation, the difference between training data and real images became evident, indicating domain shift. Images taken in real scenes differed from the curated training set in lighting, viewpoint, background clutter, and composition, and accuracy dropped as a result. This behavior is well documented in surveys of transfer learning and domain adaptation for vision and medical imaging, where models trained on a source distribution often degrade on a related but different target distribution. These reviews frame domain shift as a central deployment risk and motivate remedies that act either on the data pipeline or on the representation learned by the network [23].

Two practical strategies emerge from literature. First, data augmentation can be strengthened to better match the target distribution. One way to reduce the gap between training and real data is by using stronger augmentation. Adjustments to color, brightness, or contrast, together with small changes in angle or scale, can help models cope with variation in the field. More recent techniques such as Mixup or CutMix combine images during training and have been shown to make classifiers more robust. A survey in the Journal of Big Data reviewed these methods and reported steady improvements across image benchmarks. If the difference between source and target data is still wide, domain adaptation can be applied to narrow the gap. This feature is particularly attractive for preserving privacy on

mobile devices. For longer term resilience, domain generalization methods aim to learn features that transfer to unseen domains without target supervision and provide evaluation protocols to make that goal testable [24, 25].

5) Confidence Calibration for User Facing Decisions: Modern classifiers often miscalibrate their probabilities, meaning the reported confidence does not match empirical correctness. For a well-calibrated model, predictions made with 0.80 confidence should be correct about 80 percent of the time. This gap has been documented in large empirical studies that popularized diagnostics such as reliability diagrams and Expected Calibration Error (ECE) for classification. These works show that high-accuracy deep networks can still be over- or under-confident, so probability quality must be checked separately from accuracy [26].

A practical remedy is post hoc calibration, which adjusts the logits or probabilities of an already trained model on a small validation set. The most common tool is temperature scaling, a one-parameter transformation that improves alignment between confidence and accuracy without changing the predicted class. Temperature scaling is widely reported to reduce ECE on modern architecture and is often the first method to try in production settings. Follow-on work refines this idea with adaptive variants that learn different temperatures to cope with class imbalance, label noise, or highly accurate base models, and with local temperature scaling that targets dense prediction tasks while preserving ranking. These methods keep the model and the UI unchanged and only modify the mapping from logits to probabilities, which suits mobile deployment.

Calibration also matters in applied domains because user-facing confidence drives decisions. Medical imaging studies show that better calibrated probabilities improve downstream choices and that simple ensembling often tightens calibration further. Reviews of uncertainty in deep learning likewise recommend reporting both accuracy and calibration metrics, since confidence affects triage rules, thresholds, and when to ask for more evidence. In our app, a calibrated score would help decide when to present top-k alternatives or when to ask the user to capture another photo, and we can track ECE or related metrics alongside latency during testing [27, 28].

6) Portability and Runtime Optimizations in Practice: Keeping the model and label map as packaged assets and hiding preprocessing plus inference behind a small service layer gives you a clean boundary between ML code and the UI. This reduces coupling, makes model swaps routine, and limits long-term maintenance risk that often appears as ML “technical debt” in production systems. A modular boundary of this kind is a standard recommendation in large-scale ML engineering work.

On device, TFLite supplies the lightweight runtime needed for this pattern and is designed to run compact models across a wide range of edge targets. Its microvariant shows the same interpreter approach scaled down to tiny systems, which underscores the portability of bundling the model with the application and keeping the serving interface thin [29]. For phones specifically, performance varies by chipset and software stack, so the runtime should expose options for threads and hardware delegates. Evaluations on Android consistently report that enabling NNAPI or a GPU delegate can lift convolution throughput to real-time levels on mainstream handsets when vendor drivers are present. Community benchmarks such as MLPerf Mobile were created to measure these effects fairly across devices and software paths, which is why device-aware benchmarking belongs in the release process for any portable on device app.

Several small runtime steps help preserve responsiveness as class count or image resolution grows. Move image preprocessing off the UI thread so rendering remains smooth, then warm up the interpreter once at launch to populate kernels and caches before the first user request. After that, run the interpreter with multiple threads and turn on NNAPI or GPU delegates when the device supports them. Surveys on efficient DNN processing and independent mobile studies explain why these choices reduce latency and energy by aligning computation with the target hardware and by avoiding unpredictable network delays [30]. In short, package the .tflite and labels.txt as versioned assets, keep a narrow service layer for preprocessing and inference, and use threads plus delegates at runtime; this combination makes it easy to swap datasets and retrain for new domains while keeping the app fast on a wide range of devices [31, 32].

To clarify the originality of this work, Leafy AI can be positioned relative to prior CNN based mobile vision studies. Earlier implementations using YOLO or EfficientNet Lite primarily focused on real time detection or lightweight classification with limited datasets and partial reliance on server based processing. In contrast, the present study demonstrates a fully integrated on device pipeline that combines a large scale domain specific dataset, model optimization through TensorFlow Lite, and cross platform deployment within

a Flutter based architecture. This approach enables evaluation of the trade off between accuracy, latency, and computational efficiency under real world mobile constraints.

7) Implementation Gaps in Prior Work: From an implementation perspective, most prior efforts have followed one of two approaches: (i) training in Python with model deployment via a backend service, or (ii) embedding a lightweight CNN for a limited number of classes. While TFLite has been employed in several studies, the integration of large-class models directly within Flutter applications remains uncommon. Most of the Flutter plus AI examples are restricted to demonstrations using pre-trained models with limited scope.

Leafy AI addresses these gaps in three concrete ways:

- Scale: This database consisted of 67,200 images that included 112 ornamental plant classes. The images had a variation of lighting, perspective, and background, so the exhibited substantial visual variability. This database used a larger amount of data and a wider variation than the dataset used in mobile vision research, where the number of categories and the number of limited images were generally used. Use of more diverse training data helped models adapt themselves better and reduce the risk of overfitting.
- Integration: The workflow is end-to-end. A MobileNetV2 model is trained with transfer learning at a resolution of 224×224, converted to TensorFlow Lite (TFLite) with optional quantization, and packaged as an application asset along with the labels.txt file. Using Flutter, the app captures an image, resizes it to 224×224 pixels, and normalizes the data. Inference is executed with the TFLite interpreter, and the label with the highest probability is returned to the result screen. Assets are stored in /assets and referenced in pubspec.yaml for easier updates.
- Execution: At Android, the processing was fully done on the device, so the results could be produced in real time. Because there was no backend system and network communication, response time remained low, even on devices with mid-specifications. When supported, NNAPI delegates, GPU, or XNPACK could be used to strengthen performance. Processing on the device also strengthens privacy, reduces operating costs, and ensures that the application remains reliable in offline conditions. The model can be updated in a controlled way without changing the application architecture.

In summary, Leafy AI extends beyond prior CNN-on-mobile implementations by demonstrating a practical, cross-platform approach for deploying a large-class model on mobile devices while maintaining sufficient efficiency for everyday use.

All plant images used in this study were collected from publicly available sources or captured in field environments with institutional permission. No personal or identifiable human data were involved. The data collection process complied with institutional and research ethics standards for non human image datasets.

IV.CONCLUSION

This study demonstrates that large scale ornamental plant recognition can be executed efficiently on standard smartphones through the integration of MobileNetV2, TensorFlow Lite, and Flutter within a fully on device pipeline. The proposed system achieved 0.89 top 1 accuracy on a controlled test set while maintaining low latency and stable offline performance after optimization and quantization. Experimental comparison confirms that model conversion and integer quantization significantly reduce inference time and memory usage with only marginal accuracy degradation. Field evaluation highlights the impact of domain shift between curated and real world images, indicating the need for distribution aware training strategies. Future work will focus on stronger augmentation, domain adaptation, calibration of prediction confidence, and device specific optimization to further improve robustness and scalability.

ACKNOWLEDGEMENT

The authors acknowledge Universitas Muhammadiyah Yogyakarta for providing research funding that supported this study. The authors also thank SD Unggulan Muhammadiyah Kretek Bantul for facilitating field data collection and providing institutional support during application testing. Appreciation is extended to all participants who contributed to real world image acquisition and evaluation sessions.

REFERENCES

[1] L. N. Huynh, R. K. Balan, and Y. Lee, “DEMO: GPU-based image recognition and object detection on commodity mobile devices,” in *MobiSys 2016 Companion - Companion Publication of the 14th*

Annual International Conference on Mobile Systems, Applications, and Services, Association for Computing Machinery, Inc, Jun. 2016, p. 111. doi: 10.1145/2938559.2938577.

[2] I. Martinez-Alpiste, G. Golcarenarenji, Q. Wang, and J. M. Alcaraz-Calero, "Smartphone-based real-time object recognition architecture for portable and constrained systems," *J. Real. Time. Image Process.*, vol. 19, no. 1, pp. 103–115, Feb. 2022, doi: 10.1007/s11554-021-01164-1.

[3] V. Kimie Isuyama and B. De Carvalho Albertini, "Comparison of Convolutional Neural Network Models for Mobile Devices," 2021.

[4] J. Wu, Y. Zhang, J. Hou, W. Liu, W. Huang, and H. Bai, "PocketFlow: An Automated Framework for Compressing and Accelerating Deep Neural Networks," 2018.

[5] R. Han, Q. Zhang, C. H. Liu, G. Wang, J. Tang, and L. Y. Chen, "LegoDNN: Block-grained scaling of deep neural networks for mobile vision," in *Proceedings of the Annual International Conference on Mobile Computing and Networking, MOBICOM*, Association for Computing Machinery, 2021, pp. 406–419. doi: 10.1145/3447993.3483249.

[6] J. Choi, M. Kim, D. Ahn, T. Kim, Y. Kim, and D. Jo, "Squeezing Large-Scale Diffusion Models for Mobile," Jul. 2023, [Online]. Available: http://arxiv.org/abs/2307.01193

[7] J. Wang, B. Cao, P. S. Yu, L. Sun, W. Bao, and X. Zhu, "Deep Learning Towards Mobile Applications," Sep. 2018, [Online]. Available: http://arxiv.org/abs/1809.03559

[8] M. F. Supriadi, E. Rachmawati, and A. Arifianto, "Tampilan Pembangunan Aplikasi Mobile Pengenalan Objek Untuk Pendidikan Anak Usia Dini," *Jurnal Teknologi Informasi dan Ilmu Komputer*, no. 8, 2020.

[9] N. Luh, P. Ning, S. P. Astawa, P. Trisna, and H. Permana, "Astwana & Permana, 'AIClopedia': How Does It Facilitate Gen-Z Students in Learning English? 'AIClopedia': How Does It Facilitate Gen-Z Students in Learning English?," 2020. [Online]. Available: https://www.elitejournal.org/index.php/ELITE

[10] N. Yuniar, T. Triyaswati, P. Rizki, and A. Saputri, "Integrasi Kecerdasan Buatan (AI), Deep Learning, dan Kurikulum Berbasis Sustainable Development Goals untuk Generasi Global," Jun. 2022. [Online]. Available: https://journal.staida-sumsel.ac.id/index.php/alhaytham

[11] A. M. Atha and E. Zuliarso, "Deteksi Tanaman Herbal Khusus Untuk Penyakit Kulit Dan Penyakit Rambut Menggunakan Convolutional Neural Network (CNN) Dan Tensorflow," *Jurnal JUPITER*, no. 14, Oct. 2022.

[12] I. Y. Wulandari, N. Indroasyoko, R. Mudia Alti, Y. N. Asri, and R. Hidayat, "Pengenalan Sistem Deteksi Objek untuk Anak Usia Dini Menggunakan Pemrograman Python," *remik*, vol. 6, no. 4, pp. 664–673, Oct. 2022, doi: 10.33395/remik.v6i4.11772.

[13] M. N. Arifin, M. Umar Mansyur, A. Rahman, N. P. Dewi, F. Prasetyo, and E. Putra, "Enhanced OCR Recognition for Madurese Text Documents: A Genetic Algorithm Approach with Tesseract 5.5," *https://jurnalnasional.ump.ac.id/index.php/JUITA/*, vol. 13, pp. 109–118, 2025, Accessed: Oct. 27, 2025. [Online]. Available: https://jurnalnasional.ump.ac.id/index.php/JUITA/article/view/25794/9038

[14] H. Setyawan and D. Purbohadi, "Experimenting with AI-based mobile applications to improve student engagement in ornamental plant learning in rural Indonesian schools," *Edelweiss Applied Science and Technology*, vol. 9, no. 3, pp. 2333–2343, 2025, doi: 10.55214/25768484.v9i3.5787.

[15] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," 2018.

[16] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, and T. Weyand, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," Apr. 2017, [Online]. Available: http://arxiv.org/abs/1704.04861

[17] M. Tan and Q. V Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," 2019.

[18] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "Model Compression and Acceleration for Deep Neural Networks: The Principles, Progress, and Challenges," *IEEE Signal Process. Mag.*, vol. 35, no. 1, pp. 126–136, Jan. 2018, doi: 10.1109/MSP.2017.2765695.

[19] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge Intelligence: Paving the Last Mile of Artificial Intelligence with Edge Computing," May 2019, [Online]. Available: http://arxiv.org/abs/1905.10083

[20] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, and A. Howard, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," 2018.

[21] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, "Learned Step Size Quantization," May 2020, [Online]. Available: http://arxiv.org/abs/1902.08153

[22] Z. Dong, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "HAWQ: Hessian AWare Quantization of Neural Networks with Mixed-Precision."

[23] G. Wilson and D. J. Cook, "A Survey of Unsupervised Deep Domain Adaptation," *ACM Trans. Intell. Syst. Technol.*, vol. 11, no. 5, Sep. 2020, doi: 10.1145/3400066.

[24] A. G. Khoee, Y. Yu, and R. Feldt, "Domain generalization through meta-learning: a survey," *Artif.*

Intell. Rev., vol. 57, no. 10, Oct. 2024, doi: 10.1007/s10462-024-10922-z.

[25] C. Shorten and T. M. Khoshgoftaar, "A survey on Image Data Augmentation for Deep Learning," *J. Big Data*, vol. 6, no. 1, Dec. 2019, doi: 10.1186/s40537-019-0197-0.

[26] J. Nixon, G. Brain, M. W. Dusenberry, L. Zhang Google, G. Jerfel, and D. T. Google Brain, "Measuring Calibration in Deep Learning," 2019.

[27] A. Mehrtash, W. M. Wells, C. M. Tempany, P. Abolmaesumi, and T. Kapur, "Confidence Calibration and Predictive Uncertainty Estimation for Deep Medical Image Segmentation," *IEEE Trans. Med. Imaging*, vol. 39, no. 12, pp. 3868–3878, Dec. 2020, doi: 10.1109/TMI.2020.3006437.

[28] Z. Ding, X. Han, P. Liu, and M. Niethammer, "Local Temperature Scaling for Probability Calibration," 2021.

[29] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, and J. Li, "Tensor Flow Lite Micro: Embedded Machine Learning on TinyML System.," 2021.

[30] A. Ignatov, R. Timofte, W. Chou, K. Wang, M. Wu, and T. Hartley, "AI Benchmark: Running Deep Neural Networks on Android Smartphones," 2018. doi: https://doi.org/10.30595/juita.v13i2.25794.

[31] V. J. Reddi, D. Kanter, P. Mattson, J. Duke, T. Nguyen, and R. Chukka, "MLPerf Mobile Inference Benchmark," 2022.

[32] D. Sculley, D. Golovin, E. Davydov, T. Phillips, D. Ebner, and V. Chaudhary, "Hidden Technical Debt in Machine Learning Systems," 2015.