sensors

MDPI

Article

An Intelligent Embedded Sensing System for Real-Time Native Ecuadorian Fruit Classification on an NVIDIA Jetson Nano

Andrea Suntaxi [1], Byron Navas [1,2,*] and Julio Larco [1,3]

[1] Departamento de Eléctrica, Electrónica y Telecomunicaciones, Universidad de las Fuerzas Armadas ESPE, Sangolquí 171103, Ecuador; andreasuntaxii@gmail.com (A.S.); jclarco@espe.edu.ec (J.L.)
[2] Grupo de Investigación EmbSys, Universidad de las Fuerzas Armadas ESPE, Sangolquí 171103, Ecuador
[3] Grupo de Investigación en Sistemas Inteligentes (WiCOM-Energy), Universidad de las Fuerzas Armadas ESPE, Sangolquí 171103, Ecuador
* Correspondence: brnavas@espe.edu.ec

Abstract

The integration of artificial intelligence (AI) and edge computing into vision-based sensor systems has attracted interest for its potential to support low-latency inference at the point of sensing. However, deploying efficient classification pipelines on embedded camera platforms remains challenging because of constrained computational resources. This study presents an embedded fruit-sensing classification system built using the NVIDIA Jetson Nano, a GPU-based sensor platform. The system uses a ResNet-18 convolutional neural network trained on a dataset of 1295 images across eight classes, spanning four Ecuadorian native fruits (sweet granadilla, yellow pitahaya, tree tomato, and papaya), three common fruits (apple, banana, and tomato), and a background class, achieving per-class accuracies between 86% and 96% in live tests on the device. The model was optimized through quantization and parallel processing to reduce inference latency under continuous camera input, and a WebRTC interface streamed the classification results to a browser over HTTPS/SSL. In an exploratory session at a local fruit market, without retraining, the model identified five fruit classes in 38 of 40 placements, including banana bunches and packaged fruit. On-device evaluation showed an average GPU processing time of 17.30 ms per frame (57.79 FPS), with balanced CPU/GPU utilization and stable operation. These results show that the pipeline can support real-time fruit identification at post-harvest handling points such as market weighing stations and collection centres, and that the approach may extend to other latency-sensitive sensing tasks.

Keywords: embedded sensing; edge AI; convolutional neural network; ResNet-18; NVIDIA Jetson Nano; GPU acceleration; real-time inference; deep learning; vision-based sensor systems; smart agriculture.

Check for updates

Academic Editor: Fernando De la Prieta Pintado

Received: 11 August 2026
Revised: 8 November 2026
Accepted: 20 November 2026
Published: 26 November 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

Agriculture plays a pivotal role in Ecuador's economy, employing approximately 29% of the workforce and contributing roughly 9% to the gross domestic product (GDP) [1]. It also accounts for approximately 42% of the country's total exports [2]. Within this export-driven sector, bananas remain the primary fruit export, generating approximately USD 3.6 billion in 2023 [3], while Ecuador continues to be recognized as the world's leading banana exporter [4]. Native fruits are also gaining economic importance; for instance, pitahaya exports reached USD 171.7 million in 2023, representing a 72% year-on-year increase [5].

Despite its economic importance, agricultural production faces recurring challenges related to climate disruptions, such as El Niño events, and rising production costs [6], along with planting delays, pest pressure, soil degradation, and post-harvest losses. These challenges coexist with technological innovation and readiness gaps, reflected in Ecuador's ranking in the Global Innovation Index 2022 and the United Nations Conference on Trade and Development's (UNCTAD) Frontier Technologies Readiness Index [7,8].

To contextualize the fruit varieties examined in this study, native species occur naturally in a region without human intervention, while indigenous species may also occur naturally across multiple regions; endemic species are restricted to a specific geographic area. Ecuador and the Andean region host several such fruit species, including sweet granadilla (*Passiflora ligularis* Juss.), native to Ecuador and the Andean highlands [9]; yellow pitahaya (*Selenicereus megalanthus* Haw.), associated particularly with Ecuador's Palora region [10]; and tree tomato (*Solanum betaceum* Cav.), native to Ecuador and the Andean highlands [11]. The study also considers papaya (*Carica papaya* L.), which originated in Mesoamerica and is an indigenous fruit cultivated in Ecuador's coastal and Amazonian regions [12]. These fruits represent part of Ecuador's biodiversity and agricultural heritage and provide the regional context for the classification problem addressed in this work.

Traditional fruit classification methods have well-documented limitations, including labor-intensive manual sorting, subjective and inconsistent quality assessment, and limited scalability under resource constraints. Computer vision-based grading systems offer an alternative for improving classification accuracy and throughput [13–15]. However, recent AI-based fruit classification systems predominantly address widely traded fruits, such as apples, bananas, and citrus, and often rely on computationally intensive models that can be difficult to deploy in remote or cost-sensitive agricultural settings [16,17].

Native fruits such as sweet granadilla (grenadilla), yellow pitahaya (dragon fruit), tree tomato (tamarillo), and papaya exhibit diverse morphologies and surface textures. After harvest, they are commonly traded through collection centres and markets where produce from several species and producers converges. At these points the product type is not known in advance and must be identified or verified, for example when an operator selects the product at a weighing station or when mixed lots from smallholders are registered at intake. Public images of these fruits, such as those in the Fruits-360 dataset [18], are mostly studio captures on uniform backgrounds rather than images acquired with the camera of a deployed system, and many collection points operate with limited budgets. Embedded AI systems that execute convolutional models directly on edge devices can identify fruit at the point of handling without relying on cloud infrastructure. Species identification is also required before quality inspection, since grading and defect criteria differ between fruits.

The European HiPEAC network has identified embedded systems, mobile platforms, and data centers as strategic areas for advanced computing [19]. This identification underscores the necessity for efficient platforms that can process substantial data volumes and execute complex algorithms. Consequently, there has been a shift towards heterogeneous computing architectures, where CPUs, multiprocessors, and general-purpose graphics processing units (GPGPUs) collaborate to meet diverse computational needs [20]. Concurrently, edge computing facilitates the deployment of artificial intelligence (AI) directly on Internet of Things (IoT) and mobile devices. This approach supports on-device processing and real-time decision-making, thereby reducing reliance on cloud computing [21].

Graphics processing units (GPUs) provide highly parallel processing capabilities that have extended their use from graphics to scientific computing, machine learning, computer vision, and other compute-intensive applications. Their effective use requires specialized programming environments, such as CUDA, OpenCL, and OpenACC [22,23], as well as AI development and inference frameworks, including TensorRT, TensorFlow, and Py-

Torch [24]. In heterogeneous systems, GPUs operate alongside CPUs as parallel accelerators, enabling concurrent computation and data processing while improving the performance and energy efficiency of compute-intensive workloads [25]. These characteristics make heterogeneous CPU–GPU platforms particularly suitable for embedded AI and computer vision applications.

Convolutional neural networks (CNNs) are widely applied to image analysis tasks, such as classification, object detection, and segmentation [26]. By combining convolutional, pooling, and fully connected layers, CNNs progressively extract relevant image features [27], whereas deeper architectures can learn increasingly complex visual patterns from large datasets [28]. However, deep learning models typically require substantial computational resources for training and inference [29,30], motivating the use of hardware-accelerated platforms in embedded vision applications.

The NVIDIA Jetson Nano development kit (NVIDIA Corporation, Santa Clara, CA, USA) [31] is a compact platform described by the manufacturer as energy-efficient and designed for embedded AI workloads, including image classification, object detection, and sensor data processing. Its quad-core ARM A57 CPU and 128-core NVIDIA Maxwell GPU [32] enable local processing for edge applications, reducing dependence on cloud-based computation and its associated latency and infrastructure requirements [33]. The platform is supported by the NVIDIA JetPack SDK [34], which provides an integrated Linux-based development environment and GPU-accelerated libraries and frameworks, including CUDA, cuDNN, TensorRT, OpenCV, and PyTorch. This hardware–software integration enables the development and optimization of deep learning and computer vision applications directly on the embedded platform.

AI model development typically relies on separate training, validation, and test datasets for model fitting, hyperparameter tuning, and final evaluation of unseen data, respectively. Public image resources are available through platforms such as Kaggle, GitHub, ImageNet, Open Images Dataset V7, and Pexels, whereas frameworks such as PyTorch provide pretrained models that facilitate transfer learning.

Widely used CNN architectures for image classification include ResNet, Inception, DenseNet, and MobileNet. ResNet employs residual connections to facilitate the training of deep networks [30,35], whereas DenseNet promotes feature reuse through connections between successive layers [36]. MobileNet architectures emphasize reduced memory and computational requirements, making them particularly suitable for embedded and mobile applications [37]. Related computer vision tasks such as object detection have also benefited from architectures including SSD, Fast R-CNN, and YOLO, which offer different trade-offs between detection accuracy and computational efficiency [38–40].

Despite notable progress in artificial intelligence and edge computing, efficient fruit classification on embedded platforms remains challenging, particularly under resource-constrained and mobile conditions. Although CNNs have demonstrated promising results in agricultural applications, many existing approaches rely on high-performance or cloud-based systems, which limits their practicality for field deployment.

To address these limitations, this study proposes an AI-based embedded fruit classification system that integrates the NVIDIA Jetson Nano platform with an adapted ResNet-18 CNN. The contribution is not a new classification algorithm or neural network architecture but the design, hardware-aware optimization, and validation of an end-to-end intelligent sensing pipeline that integrates computer vision, edge AI, GPU acceleration, and real-time deployment. The system combines custom data acquisition and model training with ONNX-based conversion and TensorRT optimization, including FP16 quantization and graph-level optimization, to enable efficient inference on the resource-constrained Jetson

Nano. It also incorporates real-time visualization through WebRTC, thereby supporting cloud-independent deployment in smart agriculture scenarios.

The main contributions of this study are as follows:

- The design and validation of an end-to-end embedded intelligent sensing pipeline for fruit classification on an embedded GPU platform (NVIDIA Jetson Nano), demonstrating the feasibility of real-time inference in a laboratory setup arranged to resemble a post-harvest handling station, complemented by an exploratory session at a fruit market.
- The development of a progressive experimental methodology across three scenarios, progressing from internet-sourced images to camera-based acquisition and a final eight-class evaluation.
- The adaptation and deployment of a ResNet-18 CNN for fruit classification, achieving accuracies between 86% and 96% in live tests across native Ecuadorian fruits of agricultural relevance, including sweet granadilla (grenadilla), yellow pitahaya (dragon fruit), tree tomato (tamarillo), and papaya, as well as common fruits such as apple, banana, and tomato.
- The construction of a custom dataset following a progressive acquisition strategy: Experiment 1 used internet-sourced images as a minimum-resource baseline, while Experiments 2 and 3 relied on locally camera-acquired images, with Experiment 3 representing conditions that are more representative of real deployment environments.
- The integration of the complete deployment workflow, including ONNX export, TensorRT optimization, and embedded inference, within a unified and reproducible framework.
- The application of performance tuning and system-level optimization techniques, including model quantization and parallelization, resulting in reduced latency and improved processing efficiency on the embedded platform.

The remainder of this paper is organized as follows: Section 2 reviews the related work on fruit classification and embedded AI. Section 3 details the implementation methodology, including the Jetson Nano platform and the ResNet-18 CNN adaptation. Section 4 describes the dataset construction, hyperparameters, experimental design, and evaluation metrics. Section 5 explains the WebRTC setup used for the remote visualization. Section 6 reports the results of three experiments, including confusion matrices and per-class metrics, reports an exploratory session at a fruit market (Section 6.5), and validates the system's performance. Section 7 interprets the findings and discusses the practical implications and limitations of the study, including the intended application scenario (Section 7.5). Finally, Section 8 summarizes the contributions of this study and offers recommendations and future directions.

2. Related Work

This study focuses specifically on fruit classification and detection using embedded systems, such as the NVIDIA Jetson GPU platform, rather than disease detection, which dominates the existing literature. Table 1 presents an extended comparison of recent fruit classification and detection studies, organized by device type, model, task, software framework, and methodological approach. Although CNN-based methods are widely adopted, the analysis shows that embedded implementations remain scarce, with most research relying on general-purpose computing platforms. Several studies have employed transfer learning from pretrained models, particularly in embedded scenarios; however, few have explicitly reported the use of optimization techniques such as pruning, quantization, or deployment-specific tuning. These findings reveal that although embedded platforms are gaining interest, method-level optimization and real-time inference remain underreported, highlighting opportunities for more efficient and scalable edge-based implementations.

Table 1. Summary of recent fruit classification and detection studies with a focus on embedded deployment, highlighting platforms, models, and methods. The comparison emphasizes the limited number of works validated on embedded GPU platforms and the scarcity of studies addressing native fruit datasets under end-to-end edge deployment conditions.

Author(s)	Device	Model	Task	Target	Software	Method/Optimization
Zhang et al. [41]	Embedded	CNN	Detection	Strawberries	Python	Frame-based counting, real-time detection
Kumar & Behera [42]	Jetson NX	FFD (custom)	Detection	Aerial fruits	Python	Lightweight detection, hardware-accelerated
Ünal et al. [43]	Jetson Nano	CNN	Classification	20 fruits	Python, Keras	Transfer learning, real-time inference
Ananthanarayana et al. [44]	Jetson Nano	MobileNetV2	Classification	Apples, oranges, bananas	Python, Jetson SDK	Transfer learning, MobileNet pruning
Koirala et al. [16]	GPU workstations HPC	Faster R-CNN, SSD, YOLO	Detection, counting, yield estimation	Multiple (mango, apple, citrus, strawberry)	Python; Darknet, Caffe	Survey (custom, transfer learning)
Singh & Gill [45]	N/A	Review	Survey	Edge AI systems	N/A	Survey
Sipola et al. [46]	N/A	Review	Survey	Embedded AI	N/A	Survey
Fauziah et al. [47]	Unspecified	CNN	Classification	Tomatoes	Python	None reported
Namuche et al. [48]	Unspecified	CNN	Disease Detection	Citrus	Python	Basic CNN inference
Ji et al. [49]	PC	YOLOX	Detection	Apples	Python, YOLOX	Real-time detection with Shufflenet
Yang et al. [50]	PC	YOLOv5s (improved)	Detection	Tomatoes (multi-stage)	Python, YOLOv5	YOLOv5 enhancement for inference
Xu et al. [51]	PC	YOLOv5s	Detection	Zanthoxylum	Python, YOLOv5	YOLOv5s improvement under occlusion
Li et al. [52]	PC	YOLO (video-based)	Counting	Dragon fruit	Python, OpenCV	Frame-based counting, real-time detection
Zhang et al. [53]	PC	CNN (13-layer)	Classification	Multiple fruits	Python, TensorFlow	CNN training + augmentation
Khan & Debnath [54]	PC	CNN	Classification	Multiple fruit types	Python	Object detection and CNN tuning
Chaudhary et al. [55]	N/A	Review	Survey	Sustainable horticulture	N/A	Systematic review of AI in horticulture
Lipiński et al. [56]	GPU workstation	YOLOv8n, Faster R-CNN	Detection	Date fruits	PyTorch	Comparative detection benchmarking (mAP analysis)
Nashrulloh & Subarkah [57]	PC	EfficientNetB0	Classification	Fruit dataset (8 classes)	TensorFlow/ Keras	Transfer learning with EfficientNet architecture
Liu et al. [58]	Embedded system	RGB-D + CNN	Detection/ Sorting	Apples	ROS, vision stack	Integrated perception and robotic sorting system

Deep learning techniques for fruit classification have gained significant momentum owing to their applications in precision agriculture, automation and food supply monitoring. Numerous studies have applied convolutional neural networks (CNNs) and object detection architectures to recognize fruits from digital images. Zhang et al. [53] introduced a 13-layer CNN model with data augmentation strategies, achieving competitive accuracy under varied imaging conditions. Similarly, Khan and Debnath [54] developed a multiclass classification framework using CNNs and object detection, optimized for performance under noisy image conditions; however, their work was evaluated on desktop systems.

Numerous studies on embedded GPUs have established the viability of real-time fruit classification and detection using NVIDIA Jetson platforms. Ünal et al. [43] deployed a CNN-based system called FruitNet on a Jetson Nano for the real-time classification of 20 fruit types. Ananthanarayana et al. [44] implemented a fruit freshness classifier using MobileNetV2 on NVIDIA Jetson platforms with CMOS sensors, attaining 97% accuracy for six fruit classes. Kumar and Behera [42] introduced a fast detector achieving 100 FPS on Jetson NX, supporting autonomous aerial grasping.

In contrast, many detection-based systems have been developed on desktop or cloud-based platforms. Ji et al. [49] combined ShufflenetV2 and YOLOX to perform apple detection in real time. Yang et al. [50] improved YOLOv5 for recognizing tomato fruits at multiple growth periods, while Xu et al. [51] designed a YOLOv5s-based pipeline for detecting Zanthoxylum fruits under occlusion. Li et al. [52] also proposed a real-time video-based method to count dragon fruit flowers and fruits, although none of these were implemented on embedded devices.

Other CNN-based studies include the work of Fauziah et al. [47] on tomato classification and Zhang et al. [41], who deployed a detection system for strawberries using embedded hardware. Disease-oriented fruit recognition, such as the citrus black spot classifier by Namuche et al. [48], is more common in the literature than multiclass classification approaches for native fruits.

Several review and survey papers have contextualized these trends in the literature. Koirala et al. [16] conducted a systematic review on deep learning in fruit and plant recognition, confirming that most implementations are constrained to simulation or offline evaluation. Tang et al. [59] reviewed recognition and localization methods for vision-based fruit-picking robots and identified recognition under illumination changes and occlusion as open problems. Lin et al. [60] detected guava fruits and branches with a Mask R-CNN using a reduced backbone, trained on a small number of images, and reconstructed them in three dimensions from RGB-D data to plan collision-free paths for a harvesting robot. These picking tasks require fruit to be located in the canopy to guide a manipulator, whereas the present study addresses the identification of harvested fruit presented to a fixed camera. Singh and Gill [45] offered a broad survey of Edge AI, while Sipola et al. [46] emphasized the challenges and potential of embedded AI hardware and software platforms.

Despite the importance of this area, research on the classification of native fruits remains sparse, with even fewer studies deploying models on embedded edge devices like the Jetson Nano. MobileNet and ResNet models, which are highly optimized for embedded deployments, remain underutilized in field-ready applications. Based on the literature summarized in Table 1, most fruit classification and detection studies either rely on workstation-class hardware or focus on commercially important fruit categories. While there have been reports of embedded implementations in specific instances, we did not find any studies that explicitly focus on the classification of native Ecuadorian fruits using an end-to-end embedded GPU deployment framework. This study seeks to fill this gap by introducing an embedded classification system utilizing the Jetson Nano platform, which is optimized through lightweight convolutional neural networks (CNNs) and validated using native fruit datasets.

Recent studies from 2025 have further emphasized the growing importance of artificial intelligence in horticulture and fruit production systems. Chaudhary et al. [55] present a systematic review of AI applications in sustainable horticultural production, highlighting the integration of machine learning, Internet of Things (IoT) systems, and real-time decision support to enhance efficiency and resilience in agricultural workflows. Similarly, Lipiński et al. [56] evaluate YOLOv8n and Faster R-CNN models for date fruit detection, demonstrating high mAP values in controlled GPU-based environments, although primarily focusing on workstation-class hardware rather than edge-embedded platforms. Nashrulloh and Subarkah [57] explore EfficientNet-based fruit classification for educational datasets, emphasizing the ongoing development of lightweight CNN architectures, albeit without any embedded deployment constraints. On a broader system level, Liu et al. [58] introduced a low-cost automated apple-picking and in-field sorting system that integrates perception, robotics, and decision modules, underscoring the importance of system-level validation in real agricultural settings. While these studies illustrate rapid advancements

in fruit detection, classification, and system integration, there has been limited focus on native-fruit classification under constrained embedded GPU environments suitable for edge deployment. This observation motivated the present study.

Research Gap and Contribution Positioning

Despite significant advancements in fruit detection and classification, most existing systems rely on high-performance desktop GPUs or cloud-based infrastructures for both training and inference. Although some embedded implementations exist, they predominantly focus on widely available commercial fruits and often prioritize detection accuracy over system-level deployment constraints, such as latency, computational footprint, and integration with real-time visualization. Furthermore, native fruit varieties from biodiversity-rich regions, such as Ecuador, are underrepresented in publicly accessible datasets and embedded validation studies. The literature review identifies three primary gaps:

- Limited validation of embedded GPUs for the classification of native fruit varieties.
- A scarcity of system-level implementations that integrate optimized CNN inference with real-time streaming pipelines.
- Insufficient evaluation of classification performance under realistic, resource-constrained edge conditions.

To address these gaps, this study introduces the following:

1. The development of a curated dataset featuring native Ecuadorian fruits;
2. The adaptation and optimization of a ResNet-18 architecture for deployment on the NVIDIA Jetson Nano;
3. An end-to-end embedded pipeline that incorporates ONNX export, TensorRT optimization, and WebRTC-based streaming; and
4. Comprehensive experimental validation across progressive training scenarios.

In contrast to studies that rely solely on simulations, the proposed system emphasizes real-time embedded deployment and system integration, which are aligned with the practical requirements of edge-enabled smart agriculture.

3. Implementation

The native fruit classification system requires the integration of both hardware and software components. The hardware setup comprised a Jetson Nano development kit and a camera module. The software development involved training a CNN, specifically the ResNet-18 CNN, along with the use of NVIDIA's Jetson inference Docker container, which provides tools and libraries for AI inference. Additionally, a WebRTC server was implemented to enable remote browser-based visualization of the classification results, with all inferences executed locally on the Jetson Nano [61].

3.1. Jetson Nano Development Kit

Figure 1 shows the NVIDIA Jetson Nano System-on-Module (SoM) [62], which integrates a quad-core ARM Cortex-A57 CPU and a 128-core NVIDIA Maxwell GPU as part of the Tegra X1 series system-on-chip (SoC). This SoC also includes image signal processing (ISP) engines, memory, audio subsystems, and multiple peripheral interfaces in a compact form factor, making it well-suited for high-performance embedded-AI and vision applications.

The Jetson Nano development kit operates on a Linux-based system and supports the NVIDIA JetPack SDK for software development. As shown in Figure 2, JetPack can be used across all NVIDIA Jetson modules and developer kits owing to their shared software

stack. The SDK comprises three main components above the hardware layer. First, Jetson Linux provides a board support package (BSP) that includes the bootloader, Linux kernel, Ubuntu desktop environment, and NVIDIA drivers. Second, Jetson Platform Services offers preconfigured services designed to accelerate AI application development on Jetson devices. Third, the Jetson AI Stack is a CUDA-accelerated software stack that delivers comprehensive libraries for GPU computing, multimedia processing, graphics, and computer vision. In this study, the deployed configuration was restricted to the components required for real-time fruit classification and WebRTC streaming: the Jetson Linux (L4T) operating system, Docker-based jetson-inference runtime, and PyTorch/TensorRT/ONNX AI framework chain. The software versions on the Jetson Nano were JetPack 4.6.5 (L4T R32.7.5), CUDA 10.2.300, cuDNN 8.2.1.32, TensorRT 8.2.1, Python 3.6.9, PyTorch 1.10.0, torchvision 0.11.0a0, OpenCV 4.5.0, Docker 20.10.21, GStreamer 1.14.5, and jetson-inference (commit e8361ae) [63]. Figure 2 illustrates the work-specific stack.

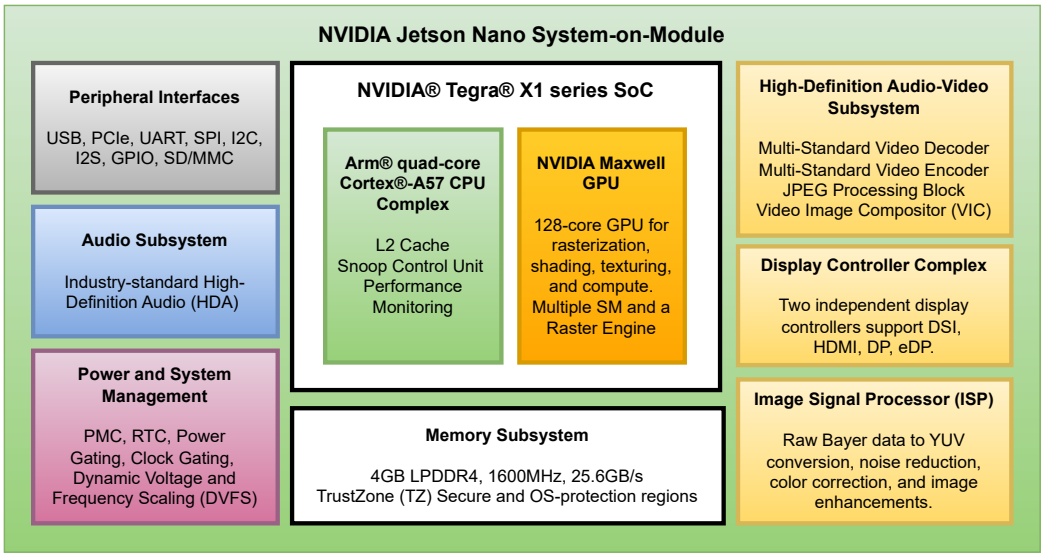


Figure 1. Overview of the NVIDIA Jetson Nano SoM architecture, highlighting the Tegra X1 SoC with CPU–GPU complex, memory, audio-video subsystems, ISP, and peripheral interfaces for embedded AI applications. Author-constructed diagram based on NVIDIA Corporation [62].

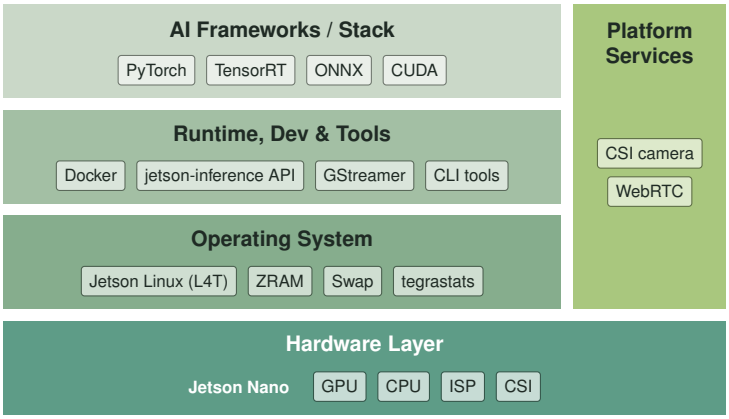


Figure 2. NVIDIA Jetson software stack architecture as configured for this work, showing the hardware layer (Jetson Nano GPU/CPU/ISP), operating system (Jetson Linux L4T), development and runtime tools (Docker, jetson-inference API, CLI tools), AI frameworks (PyTorch, TensorRT, ONNX), and deployment-relevant platform services (CSI camera, WebRTC). Author-constructed diagram based on the NVIDIA Jetson software architecture described in [64].

3.2. CNN Architecture Performance on Jetson Nano: A Literature-Based Contextual Comparison

Table 2 consolidates the quantitative performance figures for CNN architectures evaluated on NVIDIA Jetson Nano platforms, as reported in the reviewed literature. The performance varies substantially depending on the network depth, optimization strategy, and target dataset.

Prakosa et al. [65] evaluated PCENet against ResNet-18 on a cocoa-bean classification task, where PCENet achieved 86.8% Top-1 accuracy at 9.9 FPS (101 ms latency); structured pruning reduced latency to 60 ms at 16.7 FPS with a 0.7 percentage-point accuracy loss. ResNet-18 in the same study attained 82.7% accuracy at 60 ms latency (16.7 FPS). Tobiasz et al. [66] evaluated MobileNetV2 under TensorRT FP16, reporting 82.75 FPS at 12.08 ms latency; the 71.8% Top-1 accuracy figure reflects the standard ImageNet benchmark from Sandler et al. [67] and is not a Jetson-measured value. Liu et al. [68] demonstrated that ResNet-50 executes at 113.81 ms per frame (approximately 8.8 FPS) on the Jetson Nano Maxwell GPU using a unified convolution optimization framework, while MobileNet achieved 20.63 ms (approximately 48.5 FPS) under the same framework. Han et al. [69] reported LAS-RegNetY-800MF achieving 16 ms per frame through latency-aware dynamic routing. Baik et al. [70] deployed a MobileNet-based classifier on Jetson Nano at 18.43 FPS, outperforming heavier backbone combinations.

Table 2. Comparative benchmark of CNN architectures on NVIDIA Jetson platforms. All values are reported as provided in the cited references. FP32 = 32-bit floating point; FP16 = half precision; INT8 = 8-bit integer quantization; TRT = TensorRT.

Architecture	Task/Dataset	Accuracy	Latency (ms)	FPS	Platform	Optimization	Reference
Image Classification							
PCENet (original)	Cocoa bean Top-1	86.8%	101	9.9	Jetson Nano	FP32 baseline	Prakosa et al. [65]
PCENet (pruned)	Cocoa bean Top-1	86.1%	60	16.7	Jetson Nano	Structured pruning	Prakosa et al. [65]
ResNet-18	Cocoa bean Top-1	82.7%	60	16.7	Jetson Nano	FP32 baseline	Prakosa et al. [65]
MobileNetV2	ImageNet Top-1	71.8%[a]	12.08	82.75	Jetson Nano	FP16/TRT	Tobiasz et al. [66]
GhostShiftAddNet	CIFAR-10 Top-1	> GhostNet	1.3× faster (GPU)	—	Jetson Nano	Custom shift+add ops	Bi et al. [71]
GhostNet (baseline)	CIFAR-10 Top-1	—	2× slower (CPU)	—	Jetson Nano	FP32 baseline	Bi et al. [71]
ResNet-50	ImageNet Top-1	—	113.81	∼8.8	Jetson Nano	Unified GPU opt.	Liu et al. [68]
MobileNet (FADS)	Driver age/fatigue	—	—	18.43	Jetson Nano	FP32 baseline	Baik et al. [70]
LAS-RegNetY-800MF	ImageNet Top-1	—	16	—	Jetson Nano	Spatial-wise dynamic	Han et al. [69]
Jetson Xavier NX/AGX (selected results)							
ResNet-50 (Xavier NX)	Navigation/classification	—	∼8	∼125	Xavier NX	FP16/TRT	Hernandez et al. [72]
DenseNet-121 (Xavier NX)	Navigation/classification	—	∼12	∼83	Xavier NX	FP16/TRT	Hernandez et al. [72]
YOLOv5s (Xavier AGX)	Object det. COCO	37.4 mAP	∼8	∼125	AGX Xavier	FP16/TRT	Zagitov et al. [73]
This Work (Jetson Nano)							
ResNet-18 (proposed)	Native fruits dataset (8 classes)	86–96%	17.30	57.79	Jetson Nano	FP16/TRT	This work

[a] ImageNet Top-1 accuracy reported in [67]; not measured on the Jetson Nano.

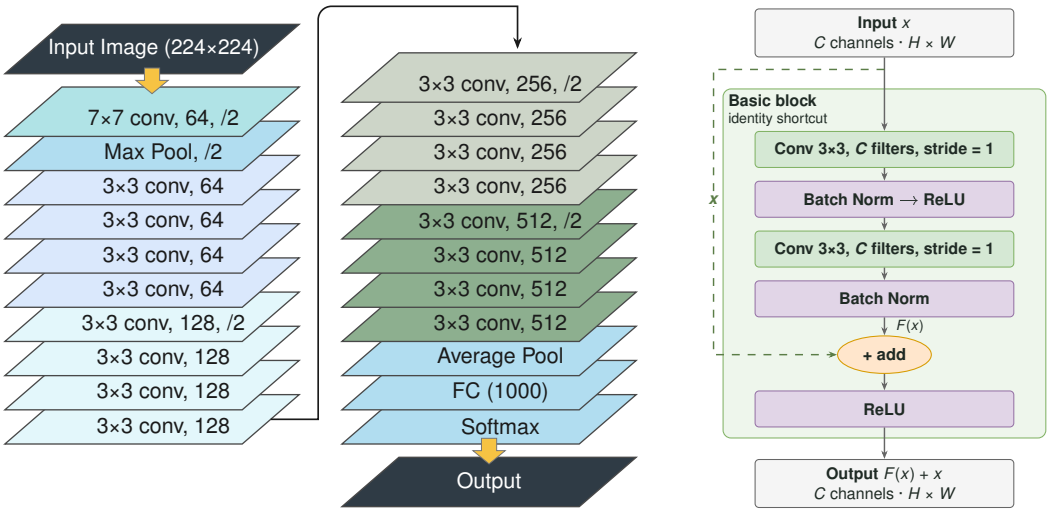
Regarding quantization, FP16 half-precision consistently delivers 1.5–2× speedup on Jetson Volta/Turing GPUs with negligible accuracy loss, while INT8 offers 2–4× gains over FP32 but may incur 1–3% accuracy degradation. TensorRT acceleration across the reviewed

corpus delivers 1.5× to 3× speedups over native PyTorch/TensorFlow inference, with the largest gains observed in the detection networks on Xavier platforms.

The literature review consolidates benchmarks from 87 distinct publications and identifies four key conclusions: (1) lightweight architectures, such as MobileNetV2, achieve real-time performance above 30 FPS on Jetson Nano with competitive accuracy; (2) TensorRT FP16 and structured pruning deliver 1.5–3× throughput improvements with less than 1% accuracy loss; (3) Jetson Xavier NX/AGX provides a 3–5× throughput advantage over Jetson Nano, although per-architecture benchmarks for Xavier remain under-reported; and (4) significant standardization gaps exist in the literature, limiting direct cross-study comparisons. In this context, the ResNet-18 implementation presented in this study achieved a per-frame GPU processing time of 17.30 ms (57.79 frames per second) in the deployed pipeline under TensorRT FP16 on Jetson Nano, with per-class accuracies between 86% and 96% in live tests on the custom native fruit dataset.

3.3. Selection and Adaptation of the Fruit Classification CNN Architecture

Several CNN architectures were considered for the native fruit classification. ResNet-18 was selected because it uses residual blocks that bypass the intermediate layers, thereby improving the model training. A key advantage of ResNet is that its residual connections mitigate the performance degradation typically observed when training very deep networks [35,61]. As illustrated in Figure 3a, the ResNet-18 CNN architecture consists of 18 layers organized in five convolutional blocks, an Average Pool layer to reduce the image dimensions, and a softmax activation layer that classifies the input into the target classes.



(**a**) (**b**)

Figure 3. Architecture of the ResNet-18 convolutional neural network (CNN) employed for native fruit classification: (**a**) overall layer organization with convolutional blocks, average pooling, and Softmax classification; (**b**) internal structure of a residual block with identity (shortcut) connection, convolutional layers, batch normalization, and ReLU activation, which enables deeper network training without degradation.

The ResNet-18 CNN architecture employs a residual block (Figure 3b) with shortcut connections that omit convolutional layers, thereby allowing direct summation of the input and output vectors. This approach prevents gradient fading and improves the training. In addition, the rectified linear unit (ReLU) activation function introduces nonlinearity while preserving computational efficiency, thereby facilitating the learning of complex patterns [74]. The basic residual block takes an input x, passes it through two 3×3 convolutional layers with their respective normalizations and ReLU activation to compute a

function $F(x)$. Subsequently, the function $F(x)$ is summed with the original input x through a direct connection, forming the final output $F(x) + x$.

To adapt the network to the fruit classification task, the final fully connected layer of the ResNet-18 CNN was modified to match the number of categories in the dataset used. Originally, this layer connected 512 inputs to 1000 output classes [35,75]. It was replaced with a configuration of (512, n), where n corresponds to the specific number of fruit classes, ensuring accurate classification, as shown in Figure 4. All layers of the adapted network were then fine-tuned with stochastic gradient descent (SGD), as detailed in Table A1 (Appendix A). Quantization strategies were also applied to reduce the model size and enhance the computational efficiency and inference speed on the Jetson Nano platform.

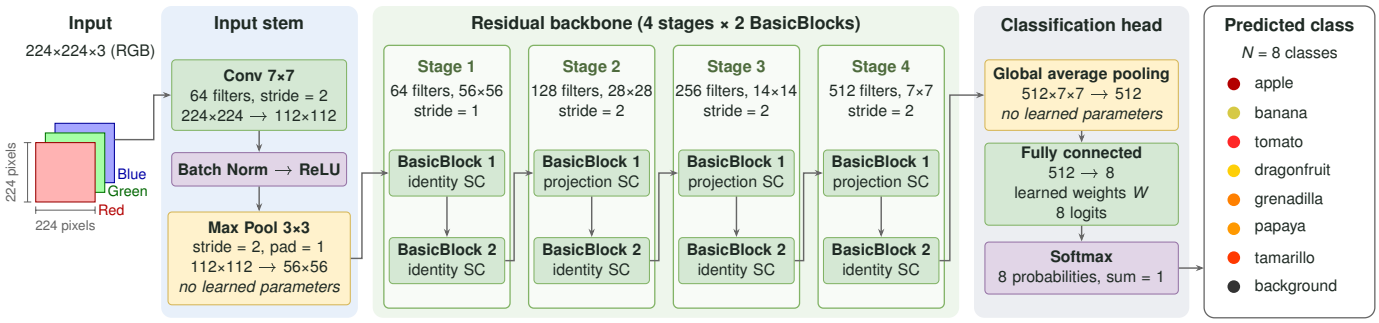


Figure 4. Adaptation of the ResNet18 CNN for fruit classification. The input stem reduces the 224 × 224 × 3 RGB image to 56 × 56 feature maps via a 7 × 7 convolution (stride = 2), Batch Normalization, ReLU, and Max Pooling. The residual backbone processes features through four stages of two BasicBlocks each, progressively doubling channel depth (64 → 512) and halving spatial resolution (56 × 56 → 7 × 7) through strided convolutions and projection shortcuts. The classification head replaces the original fully connected layer — designed for 1000 ImageNet classes — with an adapted projection (512 → N), where N = 8 represents the number of fruit categories, followed by Softmax to produce the final class probabilities. SC = shortcut connection; proj SC = projection shortcut (Conv 1 × 1, stride = 2, Batch Norm). Author-constructed figure based on [35,75]

The model selection was based on the evaluation of representative CNN architectures commonly used in image classification tasks, including Inception, MobileNet, DenseNet and ResNet. Inception achieves high accuracy but has a high computational cost because of its multi-convolutional layer structure. DenseNet enhances feature reuse but requires more memory and processing resources. MobileNet, a lightweight model, is suitable for resource-constrained devices, although it offers lower classification performance than more complex architectures. Recent advances in embedded vision have introduced lightweight architectures, such as MobileNet, EfficientNet-Lite, and ShuffleNet, which are designed to reduce the computational complexity of mobile and edge devices [76,77]. These models typically prioritize efficiency and reduced resource consumption, which may involve trade-offs in representational capacity compared with deeper model architectures.

In this study, ResNet-18 was selected as the baseline architecture owing to its well-established balance between classification performance and computational efficiency, as well as its suitability for deployment on resource-constrained platforms, such as Jetson Nano. Its residual structure enables stable training and effective transfer learning [35], which is particularly beneficial given the size and variability of the dataset.

Lightweight models are particularly suitable for resource-constrained platforms owing to their reduced memory footprint and faster inference, whereas deeper architectures may provide stronger feature representation at the cost of higher computational demand.

Notably, the primary objective of this study was not to benchmark multiple CNN architectures but to validate an end-to-end embedded AI pipeline, including data acquisition,

training, optimization using TensorRT, and real-time deployment. A direct experimental comparison with lightweight alternatives will be conducted in future studies.

3.4. Development Workflow

The overall workflow of the proposed system is organized into a sequence of interconnected stages: (1) data acquisition, (2) model training using transfer learning, (3) model optimization and export, (4) embedded deployment on Jetson Nano, and (5) real-time inference and visualization via WebRTC. These stages form a continuous pipeline, in which each step provides the input for the next, enabling end-to-end embedded classification under real-world conditions.

The overall system workflow is visually summarized in Figure 5, which complements the textual description and clarifies the relationships between the main processing stages. The process begins with platform setup by installing JetPack, which provides Linux for Tegra, the Ubuntu environment, NVIDIA drivers, CUDA, cuDNN, and TensorRT libraries. The development environment was then initialized using Docker containers that encapsulated the `jetson-inference` framework, ensuring reproducibility and simplifying dependency management. As an initial validation step, the pretrained classification and detection models included in `jetson-inference` were executed on the sample images and camera streams. The dataset preparation involved image acquisition from native fruits, annotation, and organization into training and validation subsets.

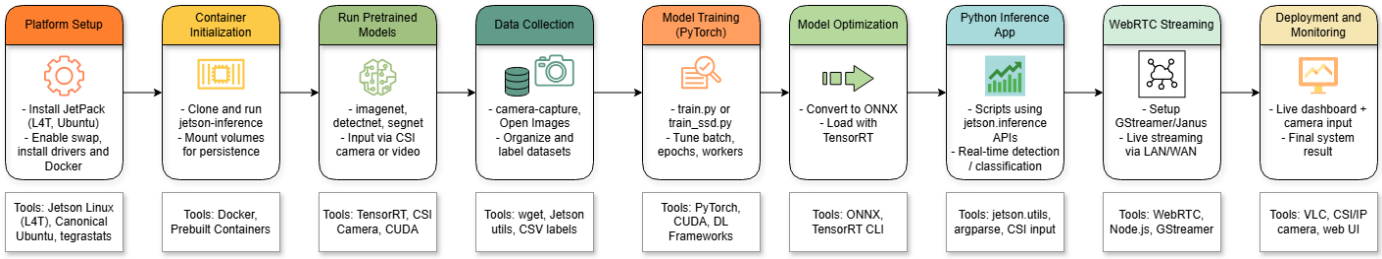


Figure 5. Development workflow for the fruit classification system on Jetson Nano, illustrating the end-to-end pipeline and relationships between processing stages. The workflow includes platform setup with JetPack, container initialization of the `jetson-inference` framework, execution of pretrained models, dataset collection and labeling, model training in PyTorch, conversion to ONNX and optimization with TensorRT, real-time inference through Python APIs, WebRTC streaming for live visualization, and final deployment and monitoring.

Model training was performed in PyTorch using transfer learning on ResNet-18, with adjustments to hyperparameters, such as the batch size, number of epochs, and number of workers, to accommodate the limited resources of the Jetson Nano. Once the training was complete, the resulting models were exported to the Open Neural Network Exchange (ONNX) format and used to generate TensorRT engines, which provided layer fusion, reduced precision (FP16), and other optimizations for efficient, real-time inference. Compared with FP32, FP16 reduces the memory footprint by approximately 50%, improving the memory efficiency and computational throughput on the Jetson Nano while preserving the classification accuracy for real-time embedded inference. This stage constitutes the inference pipeline, where the trained models are converted and executed with maximum efficiency on the Jetson GPU.

Training was carried out on the Jetson Nano using the transfer-learning tools of the jetson-inference framework. Training images are captured with the same camera and image signal processor used during inference, so training and operational images share the same domain. When a workstation or cloud GPU is available, training there is considerably faster and is the preferable option; images are still captured on the device, and the deployment

stage is unaffected, because the trained model is exported to ONNX and the TensorRT engine is always built on the target device.

It should be noted that ONNX serves exclusively as an intermediate exchange format in this pipeline; runtime inference is performed by TensorRT, not by the ONNX Runtime execution provider.

A Python-based inference application was developed using the `jetson-inference` APIs, which enabled the direct integration of the camera input and live classification output. To support remote monitoring and interactive visualization, WebRTC streaming with GStreamer/Janus was configured to transmit video streams and inference overlays over a local network Finally, dashboards and camera interfaces were used for deployment and performance monitoring, completing the development pipeline and ensuring the usability of the system under real-world conditions.

3.5. Training and Inference Workflow

While Figure 5 presents the overall development workflow of the system, Figure 6 focuses specifically on the structured five-step training and inference workflow used to adapt and deploy ResNet-18 for fruit classification. First, subfolders were created for the training, validation, and test datasets, along with a `labels.txt` file that identified each class. Subsequently, images were captured using the "Data Capture Control" tool with an 8-megapixel IMX219-77 camera (Waveshare Electronics, Shenzhen, China; Sony IMX219 sensor), ensuring proper data organization. The ResNet-18 CNN was then trained and validated using the `train.py` script, with parameter optimization based on the loss function and SGD algorithm.

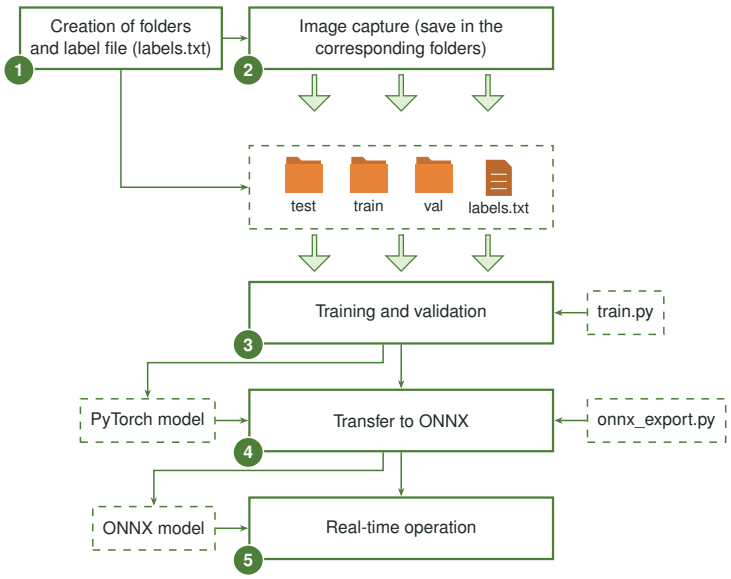


Figure 6. Five-step training and inference workflow for fruit classification: dataset folder creation and labeling, image capture, model training and validation with ResNet-18 in PyTorch, conversion to ONNX format, and deployment for real-time capable inference.

Under this approach, the system leverages a pre-trained architecture in which the initial convolutional layers reuse filters learned from large, generic databases. Starting from these pretrained weights, the final fully connected layer was replaced to match the number of classes, and all layers were fine-tuned on the custom dataset; no layers were frozen. Once the training was complete, the model was converted to the ONNX format using the `onnx_export.py` script, allowing real-time inference optimization with TensorRT. Finally,

the converted model was deployed on the system to process live images and generate fruit classification predictions.

The training code, `train.py`, was implemented in PyTorch, an open-source deep-learning framework widely adopted for computer vision applications [78]. It also incorporates `torchvision` for image preprocessing and tensor transformations and leverages cuDNN (CUDA Deep Neural Network library) to optimize GPU performance on the NVIDIA Jetson Nano platform. The hyperparameters and dataset paths were provided using the command-line arguments. Table 3 summarizes the key supporting variables, commands, functions, and classes involved in the training workflow, whereas Algorithm 1 outlines the main procedure implemented in the `main` function.

Table 3. Relevant supporting functions and code elements used in `train.py`, including the main commands, variables, functions, and classes that support the model training workflow in Algorithm 1.

Type and Description
Variable: `model_names = sorted(...)` Description: Get architecture. Gets the available network architectures.
Command: `parser.add_argument(...)` Description: Read arguments. Parses command-line arguments.
Function: `main(args)` Description: Main function. Loads the dataset, configures the model, and trains for N epochs.
Function: `train(train_loader, model, criterion, optimizer, epoch)` Description: Train model. Trains one epoch using the dataset.
Function: `validate(val_loader, model, criterion, epoch)` Description: Validate model. Evaluates model performance on the validation set.
Function: `save_checkpoint(state, is_best, ...)` Description: Save checkpoint. Saves model checkpoint during training and stores the best-performing model.
Function: `adjust_learning_rate(optimizer, epoch)` Description: Adjust learning rate. Updates the learning rate per epoch.
Function: `accuracy(output, target)` Description: Compute accuracy. Calculates prediction accuracy.
Class: `AverageMeter(object)` Description: Compute metrics. Tracks and averages metrics during training.
Class: `ProgressMeter(object)` Description: Display progress. Displays training progress metrics.

Code elements are color-coded for readability: variables and arguments in orange; functions, commands, and classes in teal. The type of element is shown in bold.

The overall training process is orchestrated by the `main` function, which manages data preparation, model configuration, and iterative optimization. The workflow begins with image normalization and the definition of preprocessing pipelines for the training and validation datasets. These datasets are then loaded using `ImageFolder` and accessed in batches through data loaders, as summarized in Algorithm 1.

A pre-trained convolutional neural network was selected and adapted to the target classification task by modifying its final layers based on the number of classes. The loss function was defined using cross-entropy, and the optimization process was performed using stochastic gradient descent (SGD). The model and loss function were transferred to the GPU to enable accelerated computation, and cuDNN benchmarking was activated to improve performance.

As shown in Algorithm 1, the training process follows an iterative loop for several epochs. In each epoch, the learning rate was adjusted, the model was trained using mini-batches from the training dataset, and validation was subsequently performed to assess generalization performance. The best validation accuracy was tracked, and the model checkpoints were stored to preserve the best-performing configuration. The training progress and performance metrics were monitored throughout the process.

Algorithm 1 Model Training (PyTorch) Main Function

```
1: Data Collection and Preparation:
    normalize ← transforms.Normalize(...)                          ▷ Adjust image pixel values
    train_transforms ← transforms.Compose([...])                   ▷ Pre-process training images
    val_transforms ← transforms.Compose([...])                     ▷ Pre-process validation images
    train_dataset ← datasets.ImageFolder(...)                      ▷ Load labeled training dataset
    val_dataset ← datasets.ImageFolder(...)                        ▷ Load labeled validation dataset
    train_loader ← torch.utils.data.DataLoader(...)                ▷ Batch access for training
    val_loader ← torch.utils.data.DataLoader(...)                  ▷ Batch access for validation
2: Model Initialization and Training Setup:
    model ← models.__dict__[args.arch](pretrained=True)            ▷ Load pre-trained model
    model ← reshape_model(model, args.arch, len(train_dataset.classes))   ▷ Adapt model to
classification task
    criterion ← nn.CrossEntropyLoss()                              ▷ Define loss function
    optimizer ← torch.optim.SGD(...)                               ▷ Initialize optimizer
    torch.cuda.set_device(args.gpu)                                ▷ Select GPU device
    model ← model.cuda(args.gpu)                                   ▷ Transfer model to GPU
    criterion ← criterion.cuda(args.gpu)                           ▷ Transfer loss to GPU
    cudnn.benchmark ← True                                         ▷ Optimize GPU performance
3: Model Training and Validation Execution:
4: for epoch in range (N) do
     adjust_learning_rate(optimizer, epoch)                        ▷ Tune learning rate per epoch
     train_loss, train_acc ← train(train_loader, model, criterion, optimizer,
epoch)                                                             ▷ Train model
     val_loss, val_acc ← validate(val_loader, model, criterion, epoch)   ▷ Evaluate model
performance
     best_accuracy ← max(val_acc, best_accuracy)                   ▷ Track best validation accuracy
     save_checkpoint({...}, is_best)                               ▷ Save model checkpoint
5: end for
```

The optimal model checkpoint was selected based on the maximum validation accuracy obtained during training. At each epoch, the validation accuracy was compared with the best recorded value using the running maximum criterion. When a higher value was obtained, the model was saved as model_best.pth.tar. When identical validation accuracy values are obtained in multiple epochs, the earliest epoch achieving that accuracy is retained, as later epochs may indicate the onset of overfitting. The selected checkpoint was subsequently used for ONNX conversion and deployment in the real-time inference system on the Jetson Nano.

During preprocessing, the input images were processed using the PyTorch's torchvision library, which provides standardized and efficient image preparation for embedded systems. The preprocessing was structured using the transforms.Compose function and integrated three essential stages. First, the camera captures, with an original resolution of 1920 × 1080 pixels, were resized to 224 × 224 pixels to meet the input resolution required by the ResNet-18 architecture. Second, the images were transformed into floating-point tensors to enable their transfer and parallel processing on the NVIDIA Jetson Nano GPU. Finally, pixel normalization was applied using the mean [0.485, 0.456, 0.406] and standard deviation [0.229, 0.224, 0.225] statistical values derived from the ImageNet dataset. This step is vital for transfer learning, ensuring that the color and brightness distributions of native Ecuadorian fruits are compatible with the filters of the initial convolutional layers of the pretrained model, thereby optimizing convergence during training and accuracy during real-time inference.

After training, the model was exported to the ONNX format, which serves as an intermediate representation to ensure interoperability between the PyTorch training environment and the deployment pipeline. The ONNX model was subsequently optimized and executed using TensorRT, which was selected as the inference backend.

TensorRT was chosen because of its tight integration with NVIDIA GPU architectures, such as the 128-core Maxwell GPU of the Jetson Nano, enabling efficient utilization of CUDA cores. Additionally, TensorRT supports FP16 quantization, reducing memory usage and improving computational performance without significantly affecting classification

accuracy. It also performs graph-level optimizations, including layer fusion and node elimination, which reduce inference latency and enhance throughput.

This optimization pipeline is particularly suitable for real-time embedded applications with limited computational resources. Alternative ONNX-compatible runtimes, such as ONNX Runtime with different execution providers, were not evaluated in this study because the focus was on a hardware-specific deployment strategy optimized for the target NVIDIA platform. Therefore, this study prioritizes a hardware-aware inference pipeline aligned with the constraints and capabilities of Jetson Nano.

3.6. Fruit Classification System Configuration

The fruit classification system was implemented using a Jetson Nano development kit connected to a display via HDMI, a keyboard via USB, and a mouse via USB. To run the model, the `jetson-inference` repository was cloned.

The training configuration involved selecting four native fruits: sweet granadilla (grenadilla), yellow pitahaya (dragon fruit), tree tomato (tamarillo), and papaya. The images were collected and organized into subfolders to create a structured dataset. Key hyperparameters, such as the number of epochs, learning rate, and batch size, were defined to optimize the model performance of the Jetson Nano.

The dataset used in the experiments consisted of images from seven fruit classes, including generic fruits such as apples, bananas, and tomatoes and native fruits such as sweet granadilla (grenadilla), yellow pitahaya (dragon fruit), tree tomato (tamarillo), and papaya. A background class was incorporated to represent the image-capturing environment. Images were collected using the "Data Capture Control" interface provided by the Jetson-inference framework (Figure 7), which enabled structured acquisition and labeling during dataset construction. The dataset comprises 1295 images (153 to 171 per class), divided into 864 training, 283 validation, and 148 test images, and is publicly available in a Zenodo repository (see Data Availability Statement). All fruit samples were commercially acquired from local markets. This study did not involve wild plant collection, protected species sampling, genetic analysis, or field harvesting; therefore, no specific permits or biodiversity compliance approvals were required.

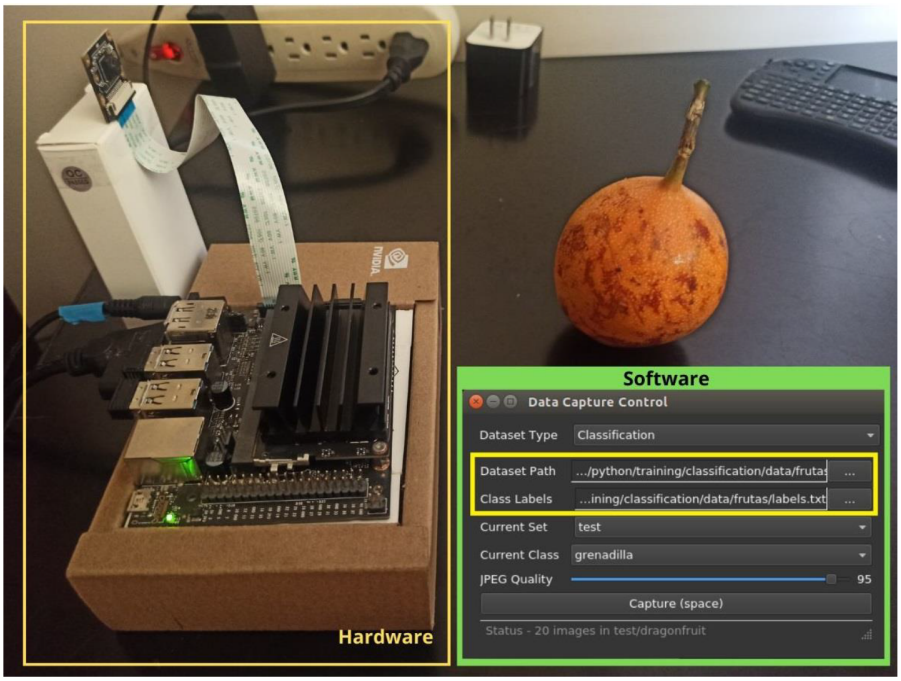


Figure 7. "Data Capture Control" interface from the jetson-inference framework, employed for structured image acquisition and class labeling during custom dataset creation.

The training process was configured using the stochastic gradient descent (SGD) optimization algorithm, which was selected for its stable convergence behavior, computational efficiency, and well-established performance in transfer learning settings. A learning rate of 0.1 and momentum of 0.9 were used, with training conducted over 85 epochs using a batch size of four. Additionally, four parallel workers were employed for data loading to improve the training throughput.

4. Custom Dataset Training and Deployment for Native Fruit Classification

The fruit classification model was trained using a proprietary dataset, following the parameters established in Section 3.6. For this purpose, the program `train.py` was executed, which trained the ResNet-18 CNN. Once this program was executed via the console, it was possible to visualize metrics such as the current epoch, loss, and precision in real-time during the training process. Once this program was executed via the console, the current epoch, loss, and accuracy could be monitored in real time during the training process. Once the training was completed, a model was obtained in the PyTorch format, which was converted to ONNX using the `onnx_export.py` program. This process generated a `resnet18.onnx` file that was stored in a specified path. Finally, real-time tests were performed by running the ONNX model, which used the label file (`labels.txt`) and specified the output of the model through a camera serial interface (CSI) camera to visualize the real-time operation [61]. The steps described above can be found in more detail in [63].

4.1. Experimental Setup

The three experiments were designed as a progressive validation framework, each employing a distinct dataset that reflected an increasing level of domain complexity rather than independent, unrelated scenarios. Experiment 1 used Internet-sourced images to establish a minimum resource training baseline. Experiment 2 introduced camera-captured images and a background class to incorporate domain realism and environmental variability. Finally, Experiment 3 employed a fully camera-acquired dataset with multiple classes, including native fruits, to simulate real-world deployment conditions. This progressive transition allows for the evaluation of the model under increasing domain complexity and ensures consistency between the dataset construction and experimental objectives.

Prior to the experiments, preliminary tests were performed to determine suitable values for the batch size and number of workers, considering the dataset size and limited resources of the Jetson Nano GPU. These tests showed that very small batch sizes reduced accuracy, whereas excessively large values and high worker counts did not yield improvements and increased training time. Based on this trade-off, batch sizes of four and four workers were selected as the default configurations for most experiments. However, in Experiment 1, a smaller configuration with a batch size of 2 and two workers was intentionally applied to validate the feasibility of training with the minimal resources.

The three experiments were designed sequentially to progressively refine and evaluate the fruit classification system under different conditions. Experiment 1 served as a baseline test using a small dataset of 150 Internet-sourced images per class, limited to three native fruits (guanábana, naranjilla, and pitahaya), with a total of 450 images. The objective was to validate the feasibility of training with minimal resources; the hyperparameters were set to `workers=2`, `batchsize=2`, and training ran for 85 epochs. Because two of these fruits are not included in the later experiments, the results of Experiment 1 are not directly comparable with those of Experiments 2 and 3.

Experiment 2 introduced improvements to address the instability observed in Experiment 1: the dataset was expanded to four camera-captured classes (apple, banana, tomato,

and a background class), with 629 images in total and the hyperparameters were increased to `workers=4` and `batchsize=4`, and English labels were adopted to facilitate portability. Finally, Experiment 3 aimed to maximize the classification capacity and diversity by including eight classes: three common fruits, four native fruits (i.e., sweet granadilla (grenadilla), yellow pitahaya (dragon fruit), tree tomato (tamarillo), and papaya), and background captured under realistic conditions, with 1295 images in total.

Dataset Preparation and Partition Strategy

Owing to the absence of publicly available datasets for native Ecuadorian fruits, a custom dataset was constructed to train and validate the proposed model in this study. This limitation is particularly relevant in regions with high agricultural biodiversity, where many fruit varieties are underrepresented in open-access image repositories. The dataset comprises 1295 images (153 to 171 per class), which can be considered limited for deep learning applications; however, the use of transfer learning with a pretrained ResNet-18 allows the model to leverage features learned from large-scale image datasets, thereby reducing the dependence on extensive labeled data. In addition, data augmentation by random resized cropping and horizontal flipping (Table A1) was applied during training to increase the dataset variability. This constraint reflects a practical challenge in deploying AI solutions for region-specific agricultural applications.

Across the three experiments, the dataset composition evolved progressively according to the objectives of each stage of the study. Experiment 1 used Internet-sourced images, whereas Experiments 2 and 3 were constructed from camera-captured images to incorporate increasingly realistic conditions. In Experiment 1, each class was divided into 105 training, 30 validation, and 15 test images. In Experiments 2 and 3, the partition is given by the training, validation, and test folders of the dataset (about 67%, 22%, and 11% of the images; Table A2), and the test images were used neither for training nor for checkpoint selection.

Each image was saved into one of the three folders at capture time with the Data Capture Control interface (Figure 7). The assignment was neither grouped by fruit specimen or capture session nor randomized, so images of one fruit captured in the same session can appear in more than one subset (Section 7.6).

Model training employed stochastic gradient descent (SGD) with momentum = 0.9, learning rate = 0.1, batch size = 4, and data-loading workers = 4, except in Experiment 1, where a smaller configuration was applied.

4.2. Experiment 1

In Experiment 1, a dataset containing 150 images per class (guanábana, naranjilla, and pitahaya) was used, with images obtained from the Internet. The hyperparameters were fixed at `workers=2` and `batchsize=2` ; training ran for 85 epochs, and metrics were reported every five epochs. As reported in Table 4, the training accuracy gradually improved, reaching its highest value of 92.70% at epoch 45, with a training loss of 0.18 and a validation accuracy of 88.89%. The best validation accuracy of 90.00% was achieved at epoch 55, after which the performance fluctuated and eventually decreased, indicating instability in convergence. The overall mean accuracy across epochs was 68.14% for training and 71.76% for validation, with standard deviations of 20.54% and 14.96%, respectively, confirming inconsistent learning. The training loss decreased sharply from 9.73 at epoch 1 to values around 0.3–0.6 after epoch 35, whereas the validation loss fluctuated more markedly, with a minimum of 0.29 but peaks above 2.6. The total training process lasted approximately 51 min (approximately 36 s per epoch). Despite achieving promising accuracy values during training and validation, the model failed to generalize to real-time classification

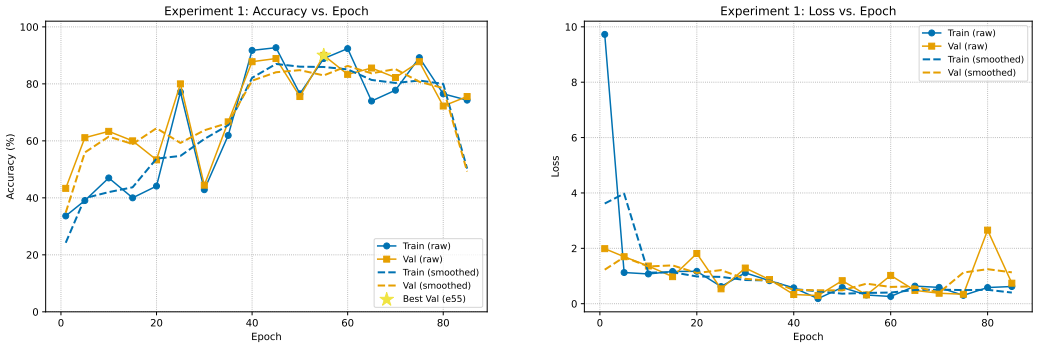
tests, suggesting possible overfitting and the need for improved hyperparameter tuning or dataset refinement [61].

Figure 8 illustrates the training dynamics of Experiment 1. The accuracy curves fluctuate markedly throughout training, with peaks near 90% and values of approximately 75% at the last epoch; the best validation performance was observed at epoch 55. The loss curves followed a complementary trend, with reductions during training and partial stabilization, although some fluctuations indicated less stable convergence than in subsequent experiments.

Table 4. Training and validation metrics across selected epochs in Experiment 1, including accuracy, loss, and training time. Summary rows report mean, standard deviation, minimum, and maximum values across epochs, highlighting convergence trends and performance stability.

Epoch	Train Loss	Train Acc. (%)	Val Loss	Val Acc. (%)	Time (min)
1	9.73	33.65	1.99	43.33	0.59
5	1.13	39.05	1.70	61.11	2.77
10	1.07	46.98	1.36	63.33	5.55
15	1.16	40.00	0.98	60.00	8.63
20	1.17	44.13	1.81	53.33	11.62
25	0.62	77.14	0.54	80.00	14.29
30	1.12	42.86	1.29	44.44	16.75
35	0.83	61.90	0.87	66.67	19.53
40	0.57	91.75	0.33	87.78	22.16
45	0.18	92.70	0.29	88.89	25.76
50	0.59	76.51	0.83	75.56	28.21
55	0.31	88.89	0.32	90.00	31.25
60	0.26	92.38	1.02	83.33	34.43
65	0.63	73.97	0.48	85.56	36.48
70	0.59	77.78	0.38	82.22	40.33
75	0.30	89.21	0.34	87.78	40.08
80	0.58	76.51	2.66	72.22	43.80
85	0.62	74.29	0.74	75.56	50.97
Mean	1.08	68.14	1.01	71.76	24.11
Std	2.10	20.54	0.60	14.96	14.52
Min	0.18	33.65	0.29	43.33	0.59
Max	9.73	92.70	2.66	90.00	50.97

Note: Summary rows report mean, standard deviation, minimum, and maximum across epochs for each metric, including training time (reported for computational efficiency and reproducibility). Shaded numbers indicate the best value for each column.



(**a**) (**b**)

Figure 8. Training metrics in Experiment 1: (**a**) accuracy curves (raw and smoothed) for training and validation, including the best validation epoch; (**b**) loss curves (raw and smoothed) for training and validation, showing convergence behavior.

4.3. Experiment 2

To address the limitations identified in Experiment 1 and improve the real-world performance, Experiment 2 incorporated several modifications to the model. A dataset of 629 images (common fruits: apple, banana, and tomato, and an additional background class) was used, with all the images captured using a camera. The hyperparameters were

set to `workers=4` and `batchsize=4` ; training ran for 85 epochs, and metrics were reported every five epochs. As shown in Table 5, the training accuracy gradually improved and reached a maximum of 93.87% at epoch 50, with a corresponding training loss of 0.18. The validation accuracy achieved 100.00% as early as epoch 40 and remained at this level in several subsequent epochs, reflecting strong convergence. The training loss decreased from 9.92 at epoch 1 to 0.16 at epoch 85, whereas the validation loss dropped below 0.3 after epoch 25 and stabilized near zero, with only minor oscillations. Across all epochs, the mean accuracies were 82.23% for training and 90.65% for validation, with standard deviations of 15.99% and 18.63%, respectively, indicating more consistent learning than in Experiment 1. The complete training process lasted approximately 1 h (approximately 43 s per epoch). Compared to Experiment 1, the introduction of a background class, adjustment of hyperparameters, and adoption of English labels improved overall performance. Nevertheless, the embedded fruit classification system still exhibited difficulties in consistently recognizing the background class [61].

Table 5. Training and validation metrics across selected epochs in Experiment 2, including accuracy, loss, and training time. Summary rows report mean, standard deviation, minimum, and maximum values across epochs, highlighting convergence trends and performance stability.

Epoch	Train Loss	Train Acc. (%)	Val Loss	Val Acc. (%)	Time (min)
1	9.92	34.43	2.02	23.19	0.72
5	1.16	54.72	0.79	68.12	3.52
10	0.73	70.05	0.29	90.58	6.78
15	0.73	72.41	0.45	78.99	10.05
20	0.51	81.37	0.20	96.38	14.06
25	0.42	84.20	0.15	99.28	16.67
30	0.41	85.38	0.13	97.10	19.97
35	0.31	88.44	0.17	94.93	25.15
40	0.24	91.04	0.04	100.00	26.65
45	0.22	92.92	0.00	99.28	31.08
50	0.18	93.87	0.02	100.00	34.01
55	0.23	92.22	0.09	96.38	39.38
60	0.23	91.98	0.05	100.00	41.99
65	0.33	87.50	0.29	88.41	44.43
70	0.25	91.27	0.15	94.93	46.17
75	0.22	93.30	0.08	97.83	49.27
80	0.38	86.08	0.06	99.28	57.13
85	0.16	93.63	0.00	100.00	60.38
Mean	0.92	82.23	0.33	90.65	29.85
Std	2.27	15.99	0.49	18.63	17.55
Min	0.16	34.43	0.00	23.19	0.72
Max	9.92	93.87	2.02	100.00	60.38

Note: Final rows summarize statistics (mean, standard deviation, minimum, and maximum) across epochs for the main performance metrics and training time. Shaded numbers indicate the best value for each column.

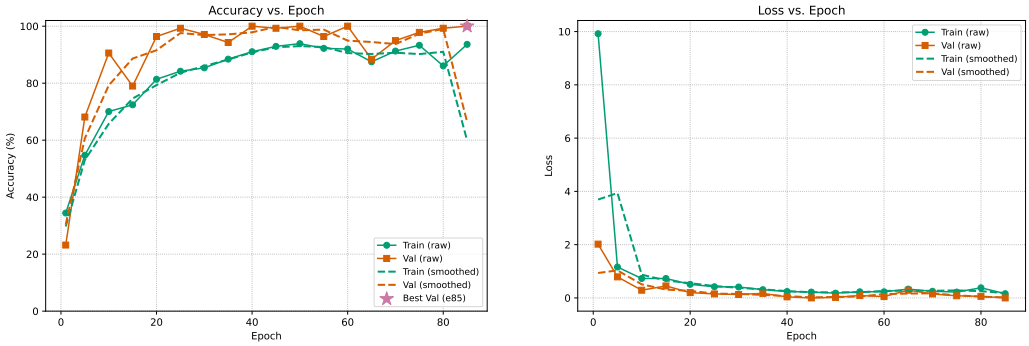
Figure 9 illustrates the learning dynamics of Experiment 2. The accuracy curves reveal a rapid increase during the early epochs, followed by convergence above 90%, with the validation accuracy reaching 100% at later stages. The loss curves show a corresponding sharp decline and subsequent stabilization, highlighting both the convergence speed and training stability across epochs.

4.4. Experiment 3

4.4.1. Training and Validation

To enhance the classification capacity and diversity, including native fruits, Experiment 3 used a dataset comprising 1295 camera-captured images. This dataset included common fruits (apple, banana, tomato), native fruits (i.e., sweet granadilla (grenadilla), yellow pitahaya (dragon fruit), tree tomato (tamarillo), and papaya), and an additional background class. The hyperparameters were set to `workers=4` and `batchsize=4` ; training

ran for 85 epochs, and metrics were reported every five epochs. The results presented in Table 6 indicate that the model achieved a training accuracy of 93.30% at epoch 85, with a training loss of 0.22 and a validation accuracy of 96.83% after 88 min of training.



(**a**) (**b**)

Figure 9. Training metrics in Experiment 2: (**a**) accuracy curves (raw and smoothed) for training and validation, including the best validation epoch; (**b**) loss curves (raw and smoothed) for training and validation, showing convergence behavior.

The model achieved a validation accuracy of 96.83% at epoch 85, which represents the best performance obtained under the current training configuration, including the selected hyperparameters, dataset size, and number of epochs used. This result corresponds to the peak validation accuracy reached after the model stabilized above 95% in the later training stages.

The summary statistics in the table further emphasize the overall convergence: the mean validation accuracy across epochs was 81.87%, with a relatively high standard deviation of 22.49%. This reflects the large gap between the early training stages and later epochs: while the validation accuracy was only 22.26% at epoch 1, it stabilized above 95% after epoch 35. This indicates that the training was unstable in the initial epochs but stabilized in later stages after convergence was achieved. The training accuracy followed a similar trend, averaging 74.01% with a peak of 93.30%. Both the training and validation losses steadily decreased and converged to values between 0.09 and 0.22 after epoch 35, indicating stable optimization despite early fluctuations. These results validate the effectiveness and consistency of the model in achieving reliable classification performance. This experiment, which included eight English-labeled classes, surpassed the previous setups in live testing.

To further assess the stability of the reported results within the available dataset, the validation accuracy was analyzed over the converged phase of training, corresponding to epochs 35 through 85. During this phase, the mean validation accuracy was 95.41% with a standard deviation of 1.26% across 11 consecutive measurement points, confirming that the reported peak validation accuracy of 96.83% represented the upper end of a consistently stable performance band rather than an isolated observation.

To assess robustness to the data partition and the random seed, a five-fold stratified cross-validation was run on the 1147 training and validation images (Table A2), without the 148 test images. Each fold used about 80% of the images for training and 20% for validation, a different fixed seed (42 plus the fold index), SGD with a learning rate of 0.1, momentum of 0.9, and weight decay of 1×10^{-4}, a batch size of 4, and 85 epochs on a cloud GPU (Google Colab, Google LLC, Mountain View, CA, USA; NVIDIA Tesla T4 GPU, CUDA 13.0, Python 3.13.16, PyTorch 2.14.1, torchvision 0.29.1); the checkpoint with the highest validation accuracy was retained. The maximum validation accuracy was 100.00% in four folds and 99.13% in the fifth (mean 99.83%, sample standard deviation 0.39 percentage

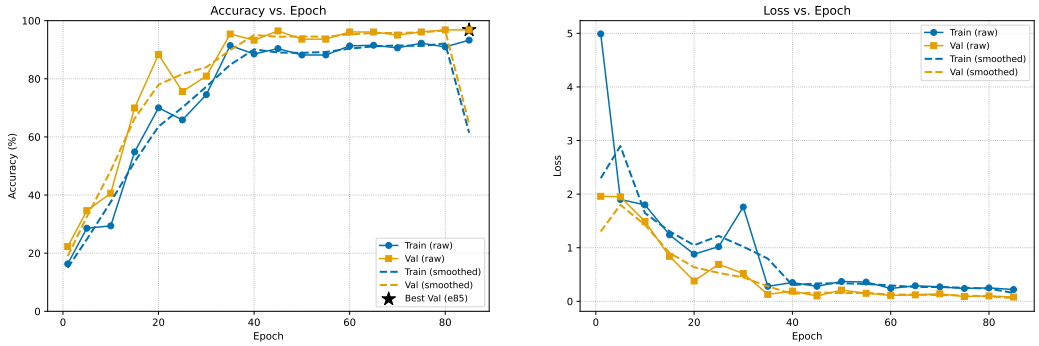
points; Table A3). Because the folds were drawn by image (Section 7.6) and each value is the best epoch of its fold, these results indicate the consistency of the training procedure rather than the accuracy on unseen fruit; they are not directly comparable with Table 6, which used the fixed folders of the dataset.

Figure 10 presents the training evolution for Experiment 3, showing the accuracy and loss across epochs for training and validation, respectively. The accuracy curves reveal a steady improvement, with the validation accuracy rising from 22.3% in the first epoch to nearly 97% by epoch 85. The best validation performance is highlighted at epoch 85, where both the training and validation accuracies plateaued above 90%. The loss curves complement this trend, decreasing sharply in the initial epochs and stabilizing below 0.3 after approximately 35 epochs, indicating a consistent convergence and robust generalization.

Table 6. Training and validation metrics across selected epochs in Experiment 3, including accuracy, loss, and training time. Summary rows report mean, standard deviation, minimum, and maximum values across epochs, highlighting convergence trends and performance stability.

Epoch	Train Loss	Train Acc. (%)	Val Loss	Val Acc. (%)	Time (min)
1	4.99	16.32	1.96	22.26	0.75
5	1.90	28.59	1.95	34.63	4.05
10	1.80	29.40	1.49	40.64	8.27
15	1.24	54.86	0.84	69.96	14.20
20	0.88	70.02	0.38	88.34	20.23
25	1.02	65.85	0.69	75.62	24.50
30	0.76	74.54	0.52	80.92	28.66
35	0.28	91.44	0.13	95.41	33.07
40	0.35	88.54	0.19	93.29	40.47
45	0.28	90.39	0.10	96.47	44.52
50	0.37	88.19	0.21	93.64	50.15
55	0.36	88.19	0.15	93.64	57.11
60	0.24	91.32	0.11	96.11	60.39
65	0.29	91.55	0.12	96.11	68.25
70	0.27	90.63	0.14	95.05	70.53
75	0.24	92.25	0.09	96.11	79.02
80	0.25	90.97	0.10	96.82	81.05
85	0.22	93.30	0.08	96.83	88.11
Mean	1.01	74.01	0.55	81.87	43.39
Std	1.09	23.45	0.51	22.49	26.76
Min	0.22	16.32	0.08	22.26	0.75
Max	4.99	93.30	1.96	96.83	88.11

Note: Mean and standard deviation values are computed across training epochs (intra-run), not across independent experimental runs. They reflect variability of metrics during the single training process. Shaded numbers indicate the best value for each column. For the converged training phase (epochs 35–85), the mean validation accuracy was 95.41% with a standard deviation of 1.26%, reflecting stable and consistent performance across the final training stages.



(**a**) (**b**)

Figure 10. Training performance in Experiment 3: (**a**) classification accuracy per epoch with best validation point marked; (**b**) loss evolution across epochs for training and validation.

4.4.2. Experiment 3: Testing

Figures 11 and 12 show examples of the trained model operating on the live camera stream. The `imagenet` program annotates each frame with the predicted class and its confidence, the score that `imagenet` reports for the top-1 class in that frame. This confidence describes a single prediction and should not be read as a probability of correct classification, so it is not a performance metric; per-class precision and recall are computed from aggregate predictions and reported in Section 6. In the frames shown for common fruits, tomato was classified with a confidence of 91.12% at approximately 40 cm from the camera, apple with 99.48% at approximately 30 cm, and banana with 99.99% close to the camera.

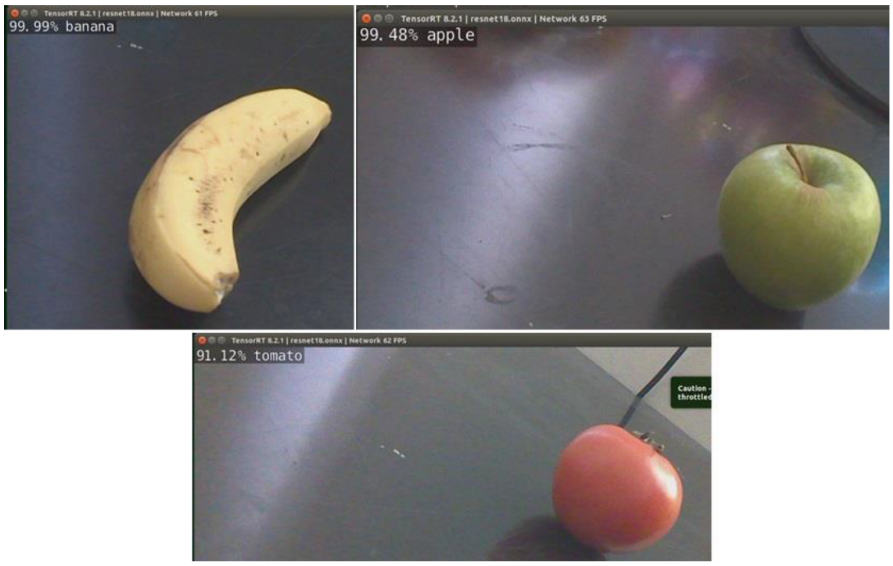


Figure 11. Examples of live classification of common fruits at different distances from the camera in Experiment 3. Each frame shows the predicted class and its top-1 confidence. Images captured on the laboratory test bench.



Figure 12. Examples of live classification of native fruits in Experiment 3, including single and multiple fruit configurations at different distances from the camera. Each frame shows the predicted class and its top-1 confidence. Images captured on the laboratory test bench.

For the native fruits (Figure 12), yellow pitahaya (dragon fruit), sweet granadilla (grenadilla), and papaya were classified with confidences of 81.51%, 83.60%, and 81.17%, respectively, when each was placed near the camera, and tree tomato (tamarillo) with 99.82% in a frame containing five fruits. These values belong to individual frames and vary from frame to frame during live operation; no conclusion about the effect of distance or of the number of fruits is drawn from them.

4.5. Data Collection Protocol

Images were captured using a standardized protocol. The camera had an 8-megapixel Sony IMX219 sensor, a field-of-view (FOV) of 77°, and 3280 × 2464 resolution. Natural daylight provided consistent lighting. Controlled neutral backgrounds and natural daylight on the laboratory test bench were used in this study. Each fruit was photographed at 0°, 45°, and 90°. The distance ranged from 20 to 50 cm. The resolution was set to 1920 × 1080 pixels and downsampled to 224 × 224 for the model input.

5. Remote Visualization

5.1. Web Server

We deployed a WebRTC server on an NVIDIA Jetson Nano platform to enable real-time remote visualization of the fruit classification results in a standard web browser. As depicted in Figure 13, the system employs an edge AI deployment model, where all CNN inferences are executed locally on the Jetson Nano GPU. Only the resulting annotated video stream was transmitted over the network to the client. This architecture eliminates the need for cloud-based processing, ensuring that raw video data remain on the edge device. At a post-harvest handling point, the device would operate at the station, while operators could supervise the results from an office or a mobile device.

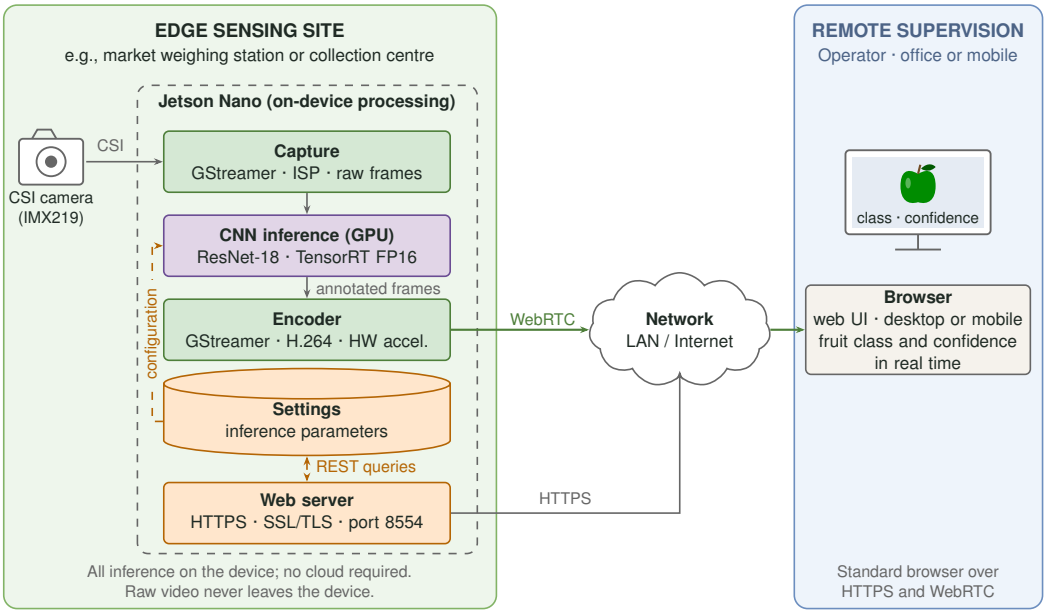


Figure 13. Edge-deployed WebRTC streaming architecture for real-time fruit classification. All CNN inference is executed locally on the NVIDIA Jetson Nano GPU. The raw video remains on the edge device; only the annotated video stream is transmitted over WebRTC, allowing for remote monitoring from any location using a standard browser.

The alternative is a remote-server implementation, in which the device only captures and encodes video and a workstation or cloud server runs the classifier. No such implementation was built in this study, but the two architectures can be compared using the

configuration of the implemented system. With the default settings of the streaming framework (H.264, 30 frames per second and a target bitrate of 4 Mbps [63]), a remote-server design requires a sustained uplink of about 4 Mbps per camera, and each classification waits for video encoding, network transfer and decoding in addition to inference. In the edge design, the classification result is a class label and a confidence value, fewer than 100 bytes per frame (below 25 kbit/s at 30 frames per second), and the annotated video is needed only for supervision. Identification therefore continues during network outages. A remote server can host larger models and centralize updates for many cameras, so it remains a valid option where a reliable network is available. The latency and bandwidth of both architectures have not been measured over the same network, and this comparison is left for future work.

WebRTC is an open-source framework for real-time audio and video communication that utilizes a peer-to-peer architecture, removing intermediaries to reduce latency and enhance transmission efficiency [79]. We integrated the WebRTC server into the jetson-inference container and utilized GStreamer-based hardware acceleration for both encoding and decoding, enabling low-latency video streaming and efficient performance, even on mobile devices [63]. The classification pipeline comprised a CSI-connected camera whose raw frames are captured through GStreamer, followed by ResNet-18 inference using the TensorRT engine, which was generated from the ONNX-exported model and executed directly on the Jetson Nano GPU. In addition, a GStreamer H.264 encoder outputs the annotated stream. Inference parameters are managed through a settings configuration store that communicates with the web server via representational state transfer (REST) queries.

The WebRTC server was configured for HTTPS with a self-signed certificate and private key generated with the `openssl` command (Algorithm 2), whose paths are passed to the server through the `SSL_KEY` and `SSL_CERT` environment variables [80]. With this configuration, the web page and the signaling messages are transmitted over TLS, and the media stream is encrypted with DTLS-SRTP, which WebRTC requires for all media [81]. A self-signed certificate does not authenticate the server: the browser cannot verify it and displays a warning that the user must accept, unless the certificate has been installed as trusted on each client. The server also does not authenticate users or restrict access, so any client that can reach its port can view the stream. The configuration therefore provides encrypted transport, but neither authenticated endpoints nor access control. In the experiments, the stream was accessed from clients on the same local network; operation over untrusted networks would require a certificate issued by a trusted authority and user authentication, which were not implemented in this study. We made the video stream accessible through the URL: `webrtc://<INTERFACE>:<PORT>/<STREAM-NAME>`, where the default port is 8554 and the stream name supports multiple concurrent streams [63].

Algorithm 2 HTTPS/SSL Setup

1: **Open the container:**
```
cd ~/jetson-inference/data
```
2: **Generate a key and self-signed certificate (valid for one year):**
```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -sha256
-days 365 -nodes -subj '/CN=localhost'
```
3: **Set the path to the SSL key (key.pem):**
```
export SSL_KEY=/jetson-inference/data/key.pem
```
4: **Set the path to the SSL certificate (cert.pem):**
```
export SSL_CERT=/jetson-inference/data/cert.pem
```

5.2. Graphical Interface

As shown in Figure 14, we visualized the fruit classification results in real time through a web browser by accessing the URL `https://<JETSON-IP>:8554`, using the `ifconfig`

command to obtain the Jetson Nano's IP address. During streaming, we monitored key performance metrics, such as the bitrate, jitter, and packet statistics (received and lost), to evaluate the transmission quality and stability [61]. This implementation demonstrates the feasibility of integrating embedded AI with real-time streaming technologies and highlights its potential applications in smart agriculture, remote monitoring, and other domains that require low latency and high computational efficiency.

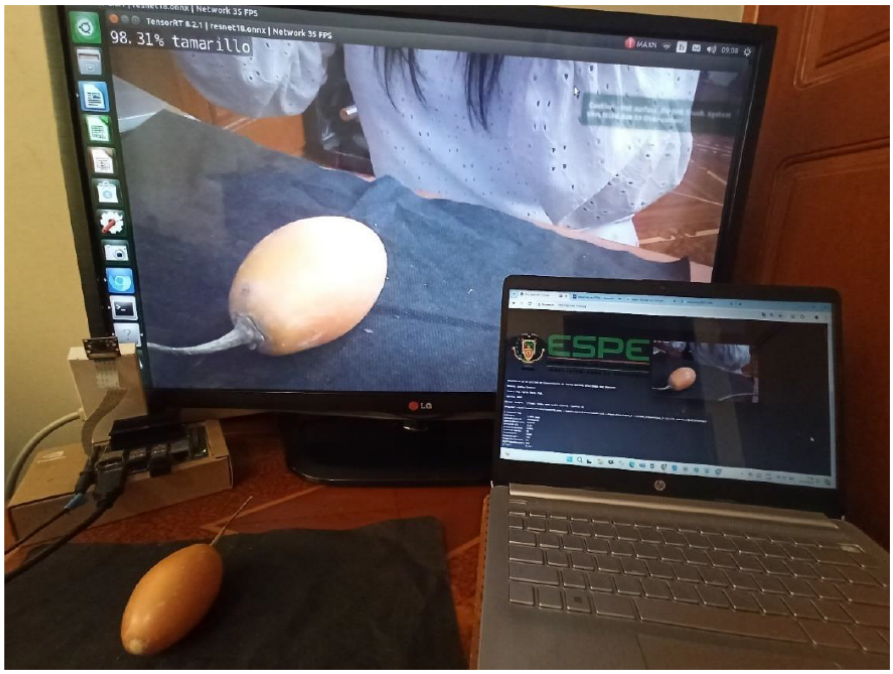


Figure 14. Experimental validation of real-time edge inference and remote visualization. A CSI camera captures a fruit sample, and the Jetson Nano performs on-device inference locally, generating annotated video with class label and confidence (e.g., tamarillo at 98.31%) displayed on a local monitor. The output is concurrently streamed via WebRTC (HTTPS) to a browser-based client, illustrating end-to-end edge processing and remote access.Laboratory setup.

6. Results

6.1. Confusion Matrix

The Experiment 3 model was evaluated in two ways. First, the checkpoint with the highest validation accuracy (epoch 54) was evaluated offline in PyTorch (FP32, validation preprocessing of Table A1) on the 148 images of the test folder (Table A2), which were used neither for training nor for checkpoint selection (Figure 15a and Table 7a). Second, the system was evaluated in live operation on the Jetson Nano (Figure 15b and Table 7b).

In each test, a fruit was presented to the camera, and the model's prediction was recorded manually and compared with the ground truth. Each fruit class was evaluated in a separate one-vs-rest session of 100 presentations, 45 of fruits of the class of interest and 55 of fruits of the other six fruit classes, for a total of 700 presentations; background scenes were not included. Five physical fruits per class were used and presented in random order, so each fruit was presented several times within and across sessions, and the presentations are not independent samples. In this context, the concepts of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) were used. For instance, for the apple class: a TP occurs when the input is an apple and the model predicts "apple"; a TN occurs when the input is another fruit and the model predicts a class other than "apple"; an FP occurs when the input is another fruit and the model predicts "apple"; and an FN occurs when the input is an apple and the model predicts a class other than "apple," such as

"grenadilla." Within each session, precision, recall, and accuracy were therefore computed as $TP/(TP+FP)$, $TP/45$, and $(TP+TN)/100$, respectively.

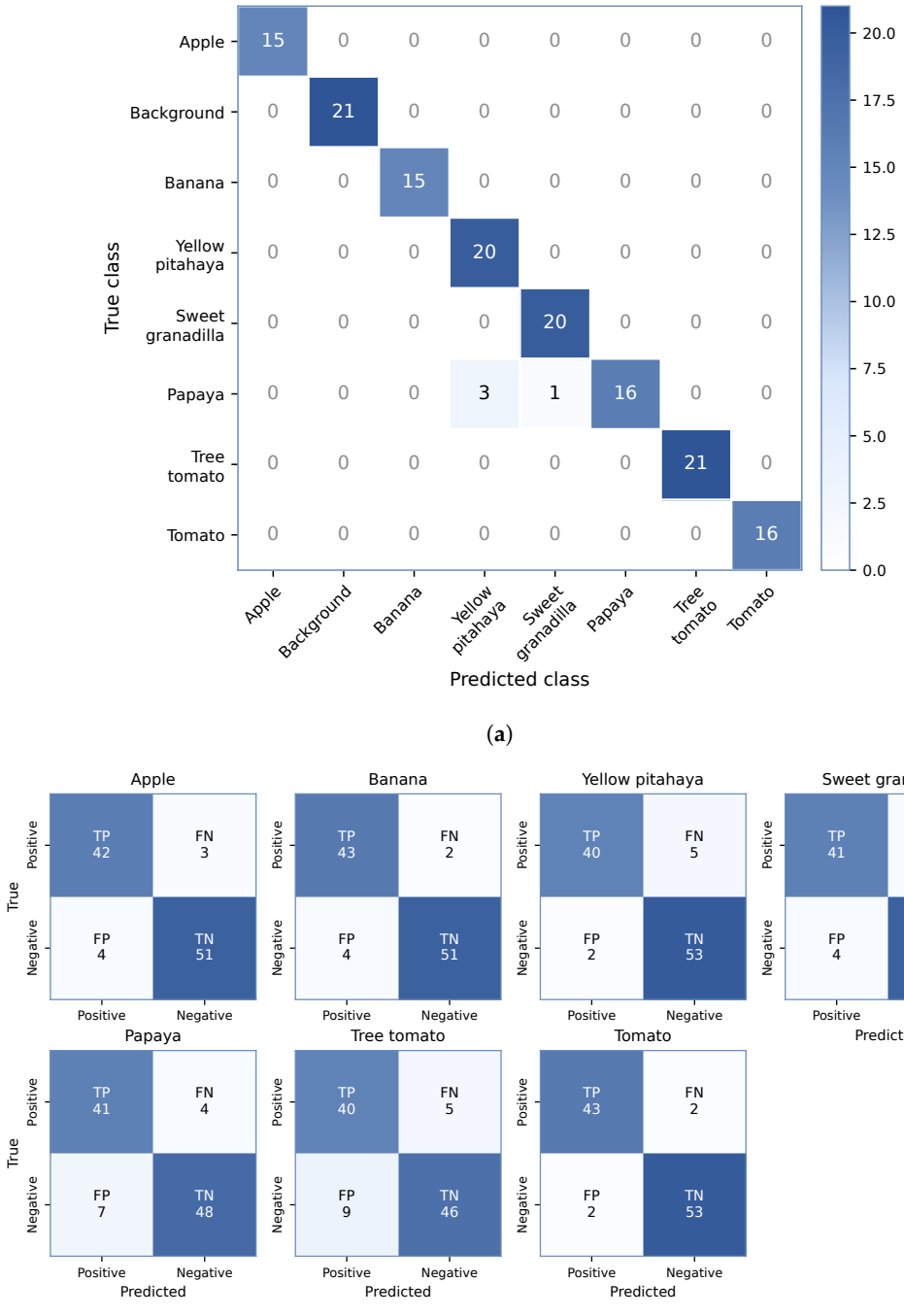


Figure 15. Confusion matrices of Experiment 3: (**a**) test set (148 images, eight classes), with rows giving the true class and columns the predicted class; (**b**) binary confusion matrix per fruit class in the live evaluation on the Jetson Nano.

The model classified 144 of the 148 test images correctly (Figure 15a), an accuracy of 97.30%. All images of seven classes, including background, were assigned to their true class. The four errors occurred in the papaya class: three images were predicted as yellow pitahaya and one as sweet granadilla. Papaya therefore has a recall of 80.00%, and yellow pitahaya and sweet granadilla have precisions of 86.96% and 95.24%, respectively. The macro-averaged precision, recall, and F1-score were 97.77%, 97.50%, and 97.43%.

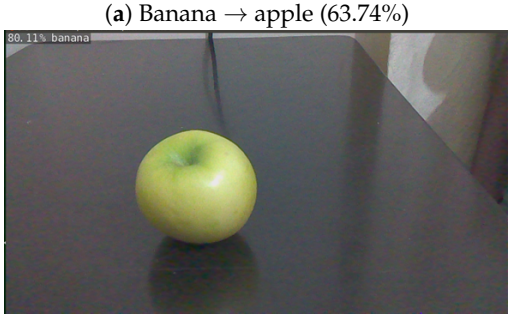
Figure 16 presents four representative misclassification examples observed during the live evaluation of Experiment 3, selected to illustrate distinct error categories. These examples confirm that classification errors are systematically associated with specific visual conditions rather than random model instability. A banana misclassified as an apple (63.74% confidence) reflects the inter-class color overlap between yellow-toned fruits. A tomato misclassified as background (44.45% confidence) illustrates fruit-to-background confusion arising from insufficient contrast between the fruit surface and the acquisition background. A green apple misclassified as a banana (80.11% confidence) represents a high-confidence error driven by the fruit's atypical green coloration, which deviates from the predominantly red-toned apple samples in the training set. Finally, a tree tomato (tamarillo) misclassified as sweet granadilla (grenadilla) (74.86% confidence) exemplifies native fruit inter-class confusion attributable to overlapping red-orange surface coloration between the two classes. These patterns are consistent with the live-evaluation results in Table 7b, confirming that classification errors are systematically associated with specific visual conditions rather than random model instability.



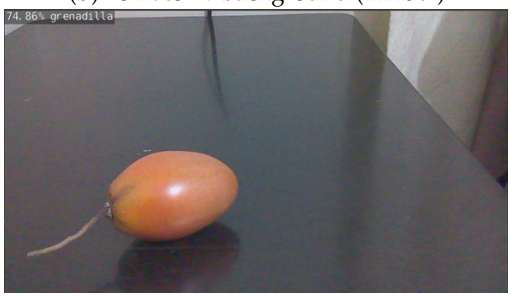
(**a**) Banana → apple (63.74%)



(**b**) Tomato → background (44.45%)



(**c**) Green apple → banana (80.11%)



(**d**) Tamarillo → grenadilla (74.86%)

Figure 16. Representative misclassification examples from Experiment 3 illustrating distinct error categories. (**a**) Banana misclassified as apple (63.74% confidence): shared yellow coloration between the two classes reduces inter-class discriminability. (**b**) Tomato misclassified as background (44.45% confidence): insufficient contrast between the fruit and the dark acquisition surface causes the scene to be assigned to the background class. (**c**) Green apple misclassified as banana (80.11% confidence): the fruit's atypical green coloration deviates from the predominantly red-toned apple training samples, producing a high-confidence erroneous prediction. (**d**) Tree tomato (tamarillo) misclassified as sweet granadilla (grenadilla) (74.86% confidence): overlapping red-orange surface coloration between native fruit classes drives inter-class confusion. In all cases, errors are attributable to specific visual conditions rather than model instability, consistent with the live-evaluation results in Table 7b. Images captured on the laboratory test bench.

Table 7. Performance metrics for each class in Experiment 3: (**a**) test set (148 images, eight classes; overall accuracy 97.30%); (**b**) live evaluation on the Jetson Nano with physical fruits (seven fruit classes).

(**a**) Test set

Class	Test Images	Precision (%)	Recall (%)	F1-Score (%)
apple	15	100.00	100.00	100.00
banana	15	100.00	100.00	100.00
yellow pitahaya	20	86.96	100.00	93.02
sweet granadilla	20	95.24	100.00	97.56
papaya	20	100.00	80.00	88.89
tree tomato	21	100.00	100.00	100.00
tomato	16	100.00	100.00	100.00
background	21	100.00	100.00	100.00
macro average	148	97.77	97.50	97.43
weighted average	148	97.59	97.30	97.23

(**b**) Live evaluation on the Jetson Nano

Class	TP	TN	FP	FN	Precision (%)	Accuracy (%)	Recall (%)	F1-Score (%)
apple	42	51	4	3	91.30	93.00	93.33	92.31
banana	43	51	4	2	91.49	94.00	95.56	93.48
yellow pitahaya	40	53	2	5	95.24	93.00	88.89	91.95
sweet granadilla	41	51	4	4	91.11	92.00	91.11	91.11
papaya	41	48	7	4	85.42	89.00	91.11	88.17
tree tomato	40	46	9	5	81.63	86.00	88.89	85.11
tomato	43	53	2	2	95.56	96.00	95.56	95.56

6.2. Performance

Table 7 lists the per-class metrics of both evaluations. On the test set (Table 7a), precision and recall were 100% for five classes, and the lowest values were the recall of papaya (80.00%) and the precision of yellow pitahaya (86.96%). In the live evaluation (Table 7b), accuracy ranged from 86% for tree tomato (tamarillo) to 96% for tomato, and the following confusions were observed.

- Tamarillo exhibited false positives, frequently misclassified as papaya or sweet granadilla (grenadilla), likely due to similar coloration.
- Banana had false positives in which it was misclassified as yellow pitahaya (dragonfruit), as both fruits share a yellow hue.
- Yellow-toned apples were often misclassified as bananas.
- Dragonfruit was occasionally misclassified as papaya, likely due to shape similarities.

Figure 17 presents a radar chart summarizing the precision and recall of the classification model per class. The results show that the model achieved a strong performance across all fruit categories, with precision and recall consistently above 80%. Notably, tomato, banana, and apple achieved the highest scores, with both precision and recall exceeding 90%, reflecting reliable detection with minimal false positives and false negatives.

Tree tomato (tamarillo) and papaya displayed comparatively lower precision, at 81.63% and 85.42%, respectively, indicating more frequent false positives for these categories. In contrast, the yellow pitahaya (dragon fruit) had a high precision (95.24%) but a low recall (88.89%), suggesting a conservative prediction pattern that tends to miss some true instances. Overall, while the model demonstrated robust classification, targeted improvements, such as class-specific data augmentation or fine-tuning, may further enhance the performance of the underperforming categories.

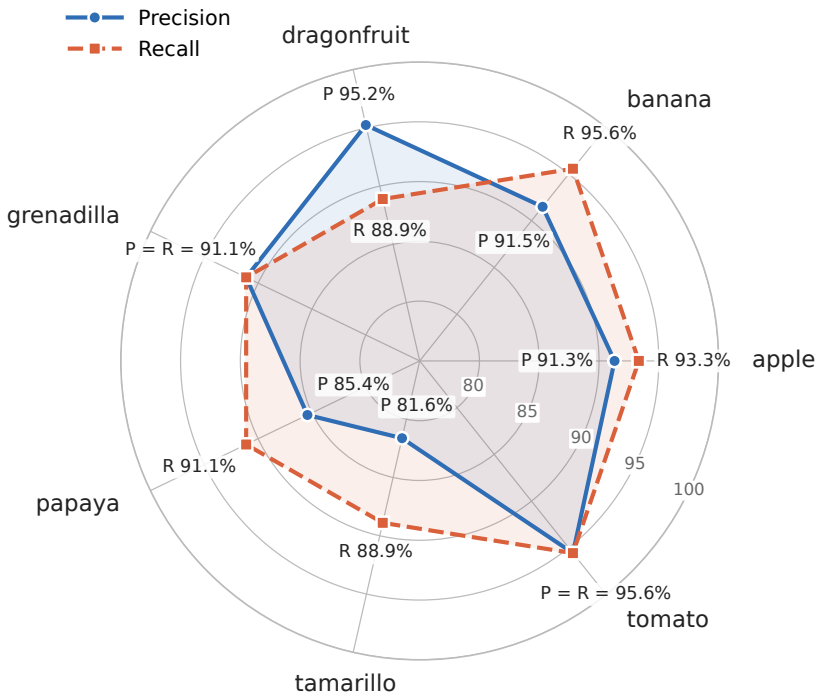


Figure 17. Experiment 3. Radar chart showing precision and recall per fruit class in the live evaluation (Table 7b), highlighting class-specific variations in performance.

6.3. Real-Time Performance Evaluation on Jetson Nano

The real-time performance of the deployed pipeline was evaluated on the Jetson Nano using the profiler of the `jetson-inference` framework and system-level monitoring with `tegrastats`. For each frame, the profiler reports the duration of the pre-processing, network inference, and post-processing stages, measured on the GPU and, separately, as host-side wall-clock time. All measurements were performed with the Jetson Nano powered through its 5 V/4 A barrel-jack input and operating in the 10 W (MAXN) power mode.

Table 8 summarizes the results. Averaged over 80 frames (ten for each of the eight classes), the GPU processing time was 17.30 ms per frame (standard deviation 1.40 ms). The TensorRT-optimized ResNet-18 network accounted for 16.70 ms of this time, whereas pre-processing and post-processing added 0.54 ms and 0.07 ms, respectively. Its reciprocal, 57.79 frames per second, expresses the processing capacity of these stages and is not a measured frame rate of the video stream. Camera capture is not included in these times, and neither are the encoding and WebRTC transmission of the annotated stream. The effect of numerical precision on the network stage is examined separately in Section 6.4.

The host-side wall-clock time for the same stages and frames was 19.00 ms per frame (standard deviation 1.53 ms; 52.62 frames per second). It describes the same GPU execution as the GPU timing. No CPU-only baseline was measured.

System monitoring indicates balanced resource utilization during operation. CPU usage remained moderate (50–55%), whereas GPU utilization reached approximately 73%, confirming that the computational workload was effectively offloaded to the GPU. Memory consumption remained stable at approximately 2.7 GB (69% of available RAM), with minimal swap usage and no evidence of memory growth or leakage. Additionally, no bottlenecks were observed in the memory bandwidth, as indicated by the EMC metrics.

Thermal measurements showed stable operating conditions, with CPU and GPU temperatures remaining at approximately 43°C and 42°C, respectively, indicating that the system can sustain continuous inference without thermal throttling.

Overall, these results demonstrate that the proposed system achieves consistent and reliable real-time inference on resource-constrained embedded platforms. This implementation maintains a balanced trade-off between latency, throughput, and resource utilization, supporting its suitability for edge-based agricultural applications. Although performance may vary depending on the input conditions and system load, the reported measurements provide a representative indication of practical deployment feasibility.

Table 8. Per-frame processing time and system resource utilization of the deployed pipeline on the Jetson Nano (FP16 TensorRT engine). Processing times are taken from the `jetson-inference` profiler and exclude camera capture, encoding, and streaming; resource values are taken from `tegrastats`.

Metric	Value	Unit	Description/Interpretation
Per-Frame Processing Time (GPU)			
Pre-processing	0.54	ms	Input preparation (negligible)
Network inference (ResNet-18)	16.70	ms	TensorRT FP16 engine; dominant stage
Post-processing	0.07	ms	Output decoding (negligible)
Total processing time	17.30	ms	Sum of the three stages
Equivalent processing rate	57.79	FPS	Reciprocal of total processing time
Host-Side Wall-Clock Time (same GPU execution)			
Total processing time (host)	19.00	ms	Same stages, measured on the host
Equivalent processing rate (host)	52.62	FPS	Reciprocal of host-side processing time
System Resource Utilization (tegrastats)			
RAM usage	2744/3964	MB	Approximately 69% usage, stable with no memory growth
SWAP usage	37	MB	Minimal usage, no memory pressure
CPU utilization	50–55	%	Moderately active, system not saturated
GPU utilization (GR3D)	73	%	Active GPU usage, effective TensorRT acceleration
Memory controller (EMC)	0	–	No memory bandwidth bottleneck observed
Thermal Behavior			
CPU temperature	43.17	°C	Stable operating condition without thermal stress
GPU temperature	42.08	°C	No thermal throttling observed

6.4. Impact of TensorRT Precision Quantization on Inference Performance

To quantify the contribution of FP16 precision quantization within the TensorRT optimization pipeline, a controlled inference comparison was conducted on the Jetson Nano using the ResNet-18 model from Experiment 3. In Experiments 1 through 3, inference was executed using the `imagenet` command provided by the `jetson-inference` framework, which applies FP16 precision by default but does not support explicit precision mode selection via command-line arguments. To enable a controlled comparison between FP32 and FP16 executions, the ONNX-exported model was compiled into two separate TensorRT engines under each precision mode using `trtexec` (NVIDIA's TensorRT engine-building and benchmarking utility). Each engine was benchmarked with `trtexec` using its default synthetic input, so the measured times do not depend on image content. Each engine was run ten times in succession on the same device; these runs are repeated measurements rather than independent replicates, and the system stability was monitored concurrently using `tegrastats`.

The values in this section describe the network stage alone, measured with `trtexec`, which reports GPU compute time and host–device transfers separately. The deployed pipeline (Section 6.3) runs the engine built by `jetson-inference` within the `imagenet` processing loop, so the absolute values in Tables 8 and 9 are not directly comparable. This section quantifies the relative effect of FP16 versus FP32 precision under identical conditions; because `imagenet` does not allow FP32 execution, no end-to-end speedup from FP16 is claimed.

The results, summarized in Table 9, demonstrate that FP16 quantization yields substantial improvements across all measured metrics relative to the FP32 baseline. The throughput increased by 106%, from 43.78 FPS to 90.03 FPS, confirming that the model processes more than twice the number of inferences per second at half precision. The mean inference latency was reduced by 52%, from 22.91 ms to 11.09 ms (standard deviation across runs, 1.68 and 0.06 ms, respectively). The difference between mean latency and GPU compute time (0.06 ms under both conditions) corresponds to the host-to-device and device-to-host data transfers, which add little to the total inference time. FP16 also gave more stable latency: the median of the per-run maximum latency decreased from 31.0 ms to 12.1 ms, and one FP32 run reached 492.5 ms.

Table 9. TensorRT inference performance comparison: FP32 vs. FP16 precision modes on the NVIDIA Jetson Nano (ResNet-18, Experiment 3 model; mean of 10 repeated runs).

Metric	FP32 (Baseline)	FP16 (Deployed)	Unit	Improvement
Inference Throughput and Latency				
Throughput	43.78	90.03	FPS	+106%
Mean inference latency	22.91 ± 1.68	11.09 ± 0.06	ms	−52%
GPU compute time	22.85	11.03	ms	−52%
Host–device transfers (H2D + D2H)	0.06	0.06	ms	—
Maximum latency (median of run maxima)	31.0	12.1	ms	−61%
Maximum latency (worst case)	83.72	12.49	ms	−85%

Note: Both conditions use the identical ResNet-18 model trained in Experiment 3 and exported to ONNX, compiled into separate TensorRT engines under FP32 and FP16 precision using `trtexec`. Inference times were measured with the default synthetic input of `trtexec`. Values describe the network stage alone, measured under conditions that differ from those of the deployed pipeline (Section 6.3); they are not directly comparable with the per-frame processing times in Table 8. Mean latency is the sum of GPU compute time and host–device transfers (H2D, host to device; D2H, device to host). Values are means over ten repeated runs of each engine on the same device; ±values are standard deviations across runs. Maximum latency is the median of the largest latency recorded in each run; the largest FP32 value, 492.5 ms, occurred in a single run. Improvement is expressed as percentage change relative to the FP32 baseline: positive values indicate higher throughput and negative values indicate lower (improved) latency. System stability was monitored via `tegrastats`; no thermal throttling or memory pressure was observed during either condition. FPS = frames per second.

6.5. Exploratory On-Site Verification at a Fruit Market

To observe the deployed system outside the laboratory, an exploratory session was carried out at a local fruit market in Sangolquí (Valle de los Chillos, Ecuador) on 5 November 2026, between approximately 19:20 and 20:15. It was a heavily rainy day after a thunderstorm, and the stall was lit by artificial light; the illuminance measured on the table with a smartphone light meter (Light Meter – Lux Meter app, Coolexp, on a Pixel 9 smartphone, Google LLC, Mountain View, CA, USA) was 361 lx. The TensorRT engine of Experiment 3 was used without retraining or any other change. A black cloth covered the market table as the station surface, and the camera, mounted on a flexible tripod, viewed it at an oblique angle, so stalls and crates of the market appear in the upper part of several frames (Figure 18a). Results were viewed on a compact 7-inch touch display (Waveshare Electronics, Shenzhen, China) connected to the Jetson Nano, which suited the limited space at the stall and a portable setup. Fruits bought at the market were placed by hand, one class at a time, as single loose fruits, as groups of the same class, and inside bags or packages, and the predicted class and its top-1 confidence were recorded from screenshots. Because the session was limited by the time available, not every class was tested in every condition.

Table 10 summarizes the 45 recorded placements. Without retraining, the model assigned apple, banana, tomato, sweet granadilla, and tree tomato to their class in 38 of 40 placements, including banana bunches, tree tomatoes in a mesh bag, and tomatoes in a

transparent bag or a factory tray covered with film (Figure 18b–e). With the hand in view, 11 of 13 placements were correct. The two errors were a sweet granadilla held by hand and one view of the tomato tray, both predicted as banana. Groups of tomatoes and the mesh bags of tree tomatoes were classified with confidences of 87.2% to 99.9%, whereas single loose tomatoes, sweet granadillas, and tree tomatoes mostly showed confidences between 41.8% and 70.0%; as noted in Section 4.4.2, these values describe single frames and are not a performance metric. Yellow pitahaya and papaya were not recognized: four placements of pitahaya were predicted as apple and one as papaya (Figure 18f), and papaya was not recognized in any attempt. The pitahayas sold at the market were at a different ripeness stage from those in the training images (compare Figure 12); with a single session, the effect of ripeness cannot be separated from that of the artificial lighting. Only a limited selection of papayas was available; the papaya attempts were not logged individually and are not included in Table 10. The empty surface was classified as background, and the network rate displayed by `imagenet` remained between 42 and 62 FPS during the session.

Table 10. Exploratory on-site verification at a fruit market: correct placements over recorded placements for each class and presentation (Experiment 3 model, no retraining). In parentheses, the placements made with the hand in view. The last column gives the range of top-1 confidence of the correct predictions.

Class	Single, Loose	Grouped, Loose	Bagged or Packaged	Total	Confidence (%)
Apple	8/8 (3/3)	–	–	8/8	69.62–100.00
Banana	4/4 (2/2)	4/4 [a]	1/1 [b]	9/9	81.73–99.98
Tomato	3/3 (1/1)	2/2	2/3 [c]	7/8	41.81–99.89
Sweet granadilla	7/8 (3/4)	–	–	7/8	43.41–92.42
Tree tomato	4/4 (2/2)	–	3/3 [d]	7/7	48.11–99.94
Yellow pitahaya	0/1 (0/1)	0/4	–	0/5	–
Total	26/28	6/10	6/7	38/45	

[a] Bunches. [b] Single banana in a transparent bag. [c] Tomato in a transparent bag (1/1) and a factory tray covered with film (1/2). [d] Several tree tomatoes in a mesh bag. Errors: sweet granadilla predicted as banana (60.63%); tomato tray predicted as banana (51.32%); yellow pitahaya predicted as apple four times (76.85–98.64%) and as papaya once (79.75%). Papaya was presented but not recognized and is not included (see text). With the hand in view, 11 of 13 placements were correct. The empty surface was classified as background (56.63%).

This session complements the laboratory evaluation; it is not a validation, since it covered one site and a small number of placements, and repeated placements of the same fruit are not independent samples. Lower confidence and class-specific errors are expected when a model trained on images from one setting is applied in another: a CNN that reached 99.35% on held-out images of plant leaves acquired under controlled conditions fell to 31.4% on images from other sources taken under different conditions [82], and illumination changes remain an open problem in fruit recognition [59]. It shows that the model, trained only on laboratory images, transferred to a real market for five of the seven fruit classes under artificial evening light, with hands in view, and with bunches, groups, and packaging. The two classes that did not transfer point to a concrete requirement for the next dataset: images captured in situ that cover the ripeness range of each fruit and the lighting of the site.

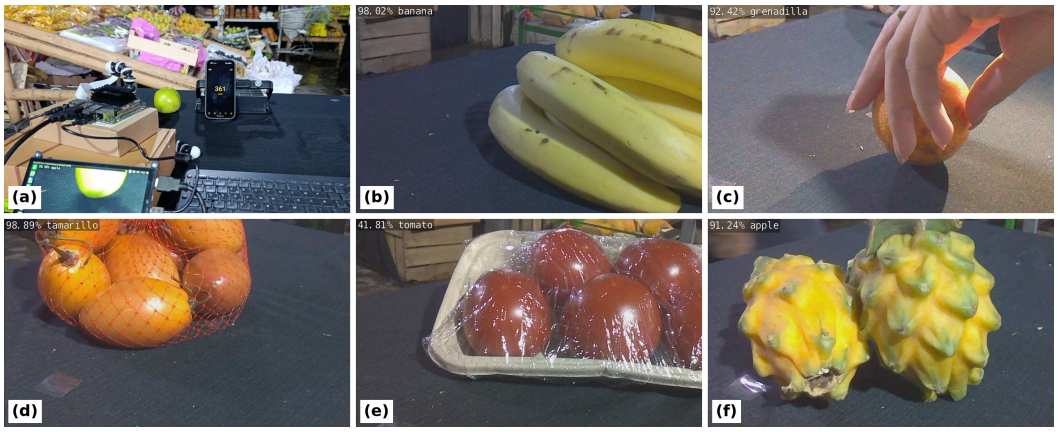


Figure 18. Exploratory on-site verification at a local fruit market in Sangolquí, Ecuador, under artificial evening light (361 lx). (**a**) Station on a market table: Jetson Nano with the CSI camera on a flexible tripod, black cloth as station surface, 7-inch touch display, and smartphone light meter. (**b**) Banana bunch, banana 98.02%. (**c**) Sweet granadilla placed by hand, grenadilla 92.42%. (**d**) Tree tomatoes in a mesh bag, tamarillo 98.89%. (**e**) Tomatoes in a factory tray covered with film, tomato 41.81%. (**f**) Yellow pitahayas misclassified as apple (91.24%). Each frame shows the predicted class and its top-1 confidence.

7. Discussion

7.1. System Feasibility and Embedded Deployment

The experimental results indicate that the proposed system can classify both native and common fruits using a continuous camera input on a resource-constrained embedded sensing platform. Our experiments confirmed the feasibility of deploying AI-based fruit classification on embedded GPUs under controlled laboratory conditions.

To bridge this gap in related studies, we validated an embedded sensing classifier for native Ecuadorian fruits using Jetson Nano. By deploying a lightweight ResNet-18 model using ONNX, TensorRT, and WebRTC, we achieved responsive embedded inference during live testing on native classes with real-time visualization. This shows that commodity edge hardware and edge AI software stacks can support local, real-time inference at the point of sensing.

From a systems perspective, the integration of quantization, ONNX export, and TensorRT acceleration reduced inference latency and enabled deployment on the Jetson Nano. In particular, the use of FP16 precision reduced memory usage and improved computational efficiency without degrading classification accuracy. This confirms the suitability of the NVIDIA JetPack SDK and PyTorch framework as deployment tools for constrained edge-sensing environments. Compared with previous studies [43,44], this study broadens the design space by incorporating the multiclass classification of native Ecuadorian fruits, a problem that remains underexplored in the literature.

The successful execution of the optimized ResNet-18 model demonstrates that embedded GPUs can sustain accurate classification without cloud reliance in controlled deployment scenarios. The integration of ONNX export and TensorRT confirms that hardware-aware optimization is critical under limited computational budgets, validating localized inference on a constrained edge-sensing device suitable for agricultural applications.

The incorporation of a WebRTC-based streaming interface extends the contribution beyond the model accuracy by enabling practical remote monitoring capabilities at the system level. This supports remote monitoring while preserving low-latency inference at the device level, which aligns with the emerging edge-computing paradigms in smart agriculture.

Overall, the findings confirm the feasibility of deploying deep-learning classifiers on embedded sensing platforms for agricultural tasks. The balance between model complexity,

training epochs, and inference latency was carefully managed, and the proposed architecture can serve as a baseline for future embedded sensing applications involving native biodiversity classification.

Real-time performance evaluation further supports the practical applicability of the proposed system in embedded scenarios with real-time constraints. The measured per-frame processing time (17.30 ms on the GPU and 19.00 ms of host wall-clock time) is well below the 33 ms frame period of a 30 fps video stream, so the processing stages fit within real-time constraints on the Jetson Nano platform. Importantly, the analysis of system-level metrics revealed balanced resource utilization, where the GPU handled the computational workload, whereas the CPU remained moderately loaded, allowing integration with additional tasks such as data acquisition or communication. The absence of memory or thermal bottlenecks also indicates that the system can sustain continuous operation under the tested conditions.

7.2. Experimental Progression, Dataset Impact, and Generalization

Across the three experiments, the performance improved as the data transitioned from Internet images to camera-captured scenes, and as the class definitions and training were refined. In the final setup (Experiment 3), the model achieved a validation accuracy of 96.83% and a test accuracy exceeding 93%, converging after approximately 35 epochs (see Figure 10 and Table 6). The 100% validation accuracy achieved in Experiment 2, despite misidentifying the background class, underscores that validation accuracy alone can be a misleading indicator of deployment-readiness. The introduction of a background class reduces false positives and improves robustness in real-world applications.

The shift from Internet-sourced images (Experiment 1) to camera-captured datasets (Experiments 2 and 3) was critical for live operation. Although Experiment 1 achieved a training accuracy of 92.70%, it failed during live testing, indicating poor domain transfer. In contrast, the models trained on camera-captured datasets operated successfully during live testing on the laboratory bench.

Regarding generalization, the experiments showed a transition from overfitting to a balanced performance. Experiment 1 achieved 92.70% training and 88.89% validation accuracy but failed during live testing, indicating overfitting to the Internet images. Experiment 2 achieved 100% validation accuracy but struggled with background identification, suggesting overfitting of the fruit characteristics. Experiment 3 achieved balanced results with 93.29% training, 96.83% validation, and > 93% test accuracy. The modest validation–test gap indicates robust generalization and confirms that validation accuracy alone is insufficient for deployment assessment. Although 150 images per class were sufficient for stable convergence and consistent validation–test results, a broader environmental evaluation would further strengthen generalization claims. The combined training, validation, and testing metrics provide insights into the reliability of the model for agricultural applications.

The progressive improvement from Experiments 1 to 3 highlights the importance of dataset curation and domain alignment when deploying CNN models in embedded-sensing environments. The initial experiment, which used heterogeneous Internet-sourced images, showed unstable generalization under real camera conditions, which is consistent with a dataset–domain mismatch reducing inference reliability in edge-based systems. In contrast, Experiments 2 and 3, which incorporated controlled camera-captured samples and native fruit classes, demonstrated improved convergence stability and validation consistency.

7.3. Quantitative Performance and Convergence Analysis

In Experiment 3, the model exhibited stable convergence, with the validation accuracy increasing from 22.3% to nearly 97% as the losses consistently decreased. After approxi-

mately 35 epochs, both the training and validation metrics plateaued, indicating a strong generalization. The mean validation accuracy of 81.2% (±23.9) across epochs reflected an improvement from the early to late stages. The final accuracy matched or exceeded those reported in related works, with classification accuracies in the 90–95% range [14,43], and in some cases, approached 97% [44]. These results underscore the robustness and competitiveness of the proposed approach for native fruit classification on embedded sensing systems, which is consistent with the broader adoption of deep learning techniques in fruit detection and agricultural vision systems [16].

7.4. Class-Wise Analysis and Environmental Context

On the test set (Figure 15a and Table 7a), all errors involved papaya images assigned to yellow pitahaya or sweet granadilla. In the live evaluation (Figure 15b and Table 7b), precision and recall were at or above 80% across all fruit classes. The accuracies of tomato, banana, and apple surpassed 90%, whereas those of papaya and tree tomato were lower owing to color and shape overlap. The radar chart (Figure 17) highlights the underperforming classes, indicating where additional data or class-specific tuning could be advantageous.

The inclusion of environmental context, namely the background class in Experiments 2 and 3, reduced false positives and enhanced robustness under real-world conditions. Despite Experiment 2 achieving 100% validation accuracy, it failed to identify the background class. Additionally, the use of English labels in Experiment 3 reduced class ambiguity by enforcing stricter data structuring during collection and preprocessing.

The live evaluation highlighted key classification trends. Classes such as tomato and banana achieved high precision and recall, whereas tree tomato (tamarillo) showed lower performance, with precisions and recalls of 81.63% and 88.89%, respectively. Misclassifications were largely due to similarities in color and shape among fruits. These findings indicate that, although the ResNet-18 CNN backbone is effective for visual categorization, further enhancement may require data augmentation or feature separation strategies, such as attention mechanisms or ensemble learning.

7.5. Application Scenario and Scope

The results should be read in relation to the intended application. The system targets post-harvest settings in which the product type is unknown or must be verified; in-field recognition is excluded because the grower already knows the cultivated species. Figure 19 places the system within the post-harvest chain.

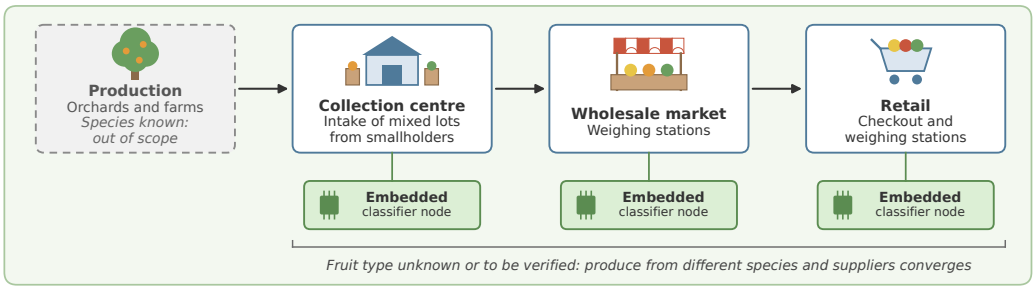


Figure 19. Application scenario of the proposed embedded classifier within the post-harvest chain. Fruit type is identified where produce from different species and suppliers converges, such as collection centre intake and market weighing stations; production is outside the scope because the cultivated species is known.

Two operational settings match the evaluated configuration closely. At weighing and checkout stations in wholesale and retail markets, fruit is placed on a known surface in front of a fixed camera, and the background class corresponds to an empty station. At collection

centres, automatic identification during intake can record the product delivered by each supplier and support traceability without manual data entry. Species identification also precedes any species-specific quality inspection, which is outside the scope of this study. Because inference runs locally on the Jetson Nano, identification continues during network outages, while the WebRTC interface allows remote supervision whenever a connection is available.

What this study validates is the sensing and classification layer, under controlled conditions that resemble such stations. An exploratory session at a fruit market (Section 6.5) showed the system operating under real retail conditions, but sustained operation in markets or collection centres, with continuous handling and throughput, has not yet been evaluated and is the next stage of this work.

7.6. Limitations and Future Work

Despite these encouraging results, the proposed approach has some limitations. First, all images were acquired at a single site under natural daylight, with fruit placed 20 to 50 cm from a fixed camera and purchased from local markets without recording its ripeness stage or cultivar. Robustness to other illumination conditions, partial occlusion, ripeness stages, cultivars, and acquisition sites was therefore not evaluated. Second, although the training time and performance were optimized for Jetson Nano, challenges may arise when scaling to additional fruit classes or deploying on lower-end hardware. A further practical consideration for long-term deployment is performance drift arising from environmental variability, such as changing lighting conditions and changes in fruit surface appearance across ripeness stages, which would require periodic monitoring and retraining in a production setting.

The models were trained and evaluated on a laboratory test bench arranged to resemble the geometry of a weighing or intake station. The on-site session (Section 6.5) extended the evaluation to artificial evening light, hands in the field of view, bunches, and packaging, but it covered one site, a short time, and a small number of placements, so classification accuracy under operational conditions has not yet been measured. The session also showed that recognition of yellow pitahaya and papaya depends on how well the training images cover the ripeness range of these fruits and the lighting of the site. In addition, the system is a single-label image classifier and assumes one product type per weighing; mixed lots of several species in the same frame would require an object detector. A pilot evaluation at a post-harvest handling point, with images captured in situ, is the next step before the results can be generalized to operational use.

The reported test accuracy may also overestimate the accuracy on fruit not seen during training, because the subsets were not separated by fruit specimen or capture session. An automatic perceptual-similarity comparison of the 1295 images identified 31 closely similar pairs across subsets. On visual inspection, two pairs were apparently identical photographs, consecutive captures of an unmoved fruit; at most one of their images is in the test set, so they cannot lower the test accuracy by more than 0.02 percentage points. The other pairs showed the same fruit from different viewpoints or different fruits on the same background. The effect of shared specimens and capture sessions cannot be estimated from the present split, so leakage between subsets is not ruled out. Future evaluations should assign each specimen and capture session to a single subset and report the resulting split.

To overcome these limitations, future research could incorporate data augmentation techniques, such as rotation and brightness jittering, expand the dataset with fruit of other ripeness stages and cultivars from diverse geographic regions, and explore hybrid models that integrate convolutional neural networks with lightweight transformer modules.

Future work could also include a comparative evaluation with lightweight architectures, such as MobileNet, EfficientNet-Lite, and ShuffleNet, to further analyze the trade-offs between accuracy, computational cost, and real-time performance in embedded deployment scenarios.

Regarding statistical validation, the five-fold cross-validation in Table A3 drew its folds by image; a cross-validation grouped by fruit or capture session and evaluation of external datasets would provide stronger generalization guarantees than the holdout evaluation reported in this study. These directions are identified as priorities for future work, particularly as publicly available datasets for native Ecuadorian fruit varieties become accessible through ongoing regional biodiversity documentation efforts.

8. Conclusions

This study demonstrated the feasibility of implementing an embedded intelligent sensing system for native fruit classification on an NVIDIA Jetson Nano with continuous camera input, integrating deep learning techniques with edge computing. By adapting a ResNet-18 architecture and training it on a dataset of native and common Ecuadorian fruits, the embedded system achieved classification accuracies between 86% and 96% in live tests, confirming its suitability for resource-constrained sensing environments.

Model optimization strategies, including quantization and parallelization, enhance the inference efficiency. The deployment pipeline, which comprises ONNX export, TensorRT optimization, and WebRTC-based streaming, establishes a complete end-to-end embedded sensing framework for low-latency classification and remote monitoring.

The results, particularly those of Experiment 3, emphasize the importance of dataset composition, label consistency, and the inclusion of environmental classes for a robust real-world performance. Progressive improvements across experiments highlight the impact of dataset–domain alignment on generalization in embedded AI sensing systems. Unlike workstation-dependent approaches, the proposed solution validates localized inference without relying on cloud infrastructure, supporting low-latency and connectivity-independent sensing for agricultural applications.

In addition, real-time performance analysis demonstrated that the proposed implementation can achieve reliable inference on resource-constrained embedded sensing platforms. The system maintained low latency and stable operation while effectively leveraging GPU acceleration through TensorRT, confirming that the deployment pipeline is suitable for edge-sensing applications. These results validate the classification performance and real-time operation under the tested laboratory conditions; integration of the system into operational post-harvest settings has not yet been evaluated. In an exploratory session at a fruit market, five fruit classes were identified under artificial evening light, with hands and packaging in view, and the results for yellow pitahaya and papaya identified the ripeness range and lighting covered by the training images as the main requirements for field deployment.

Overall, the primary contribution of this study lies in the system-level integration, progressive experimental validation, and embedded deployment of an intelligent sensing pipeline, rather than the proposal of a novel network architecture.

Future research should include a pilot evaluation at a collection centre or market station and expand the dataset under diverse illumination conditions, with images captured in situ that cover the ripeness range of each fruit and common retail packaging, and with additional native fruit varieties. Power consumption and energy per inference should also be measured, since they were not characterized in this study. The full image collection supporting this dataset is openly available (see Data Availability Statement), providing a foundation for future expansion and re-use. Overall, these results show that the pipeline

can support real-time fruit identification at post-harvest handling points such as market weighing stations and collection centres, and that the approach may extend to other latency-sensitive sensing tasks.

Author Contributions: Conceptualization, B.N. and A.S.; methodology, A.S. and B.N.; software, A.S.; validation, A.S., B.N. and J.L.; formal analysis, A.S.; investigation, A.S. and B.N.; resources, A.S. and B.N.; data curation, A.S. and B.N.; writing—original draft preparation, A.S. and B.N.; writing—review and editing, A.S., B.N. and J.L.; visualization, A.S. and B.N.; supervision, B.N.; project administration, B.N.; funding acquisition, B.N. and J.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Universidad de las Fuerzas Armadas ESPE grant number 2020-PIC-013-CTE. The APC was funded by Universidad de las Fuerzas Armadas ESPE.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The dataset presented in this study is available in the Zenodo repository at https://doi.org/10.5281/zenodo.21839051. It contains the 1295 images used in Experiments 2 and 3, in the training, validation, and test folders listed in Table A2; the Internet images of Experiment 1 are not redistributed. The dataset is openly accessible.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Appendix A. Training Protocol and Dataset Composition

Table A1. Training protocol of Experiments 1 to 3, run with the `train.py` script of the `jetson-inference` framework [63] (PyTorch) on the GPU of the Jetson Nano Developer Kit (4 GB).

Item	Setting
Initialization	ResNet-18 pretrained on ImageNet; final fully connected layer replaced by a $512 \times n$ layer ($n = 3$, 4, and 8 in Experiments 1, 2, and 3)
Trainable layers	All layers
Input	RGB, 224×224 pixels, ImageNet mean and standard deviation normalization
Training augmentation	Random resized crop (area scale 0.08 to 1.0, aspect ratio 3/4 to 4/3) and random horizontal flip ($p = 0.5$)
Validation preprocessing	Shorter side resized to 224 pixels and center crop to 224×224
Loss function	Cross-entropy
Optimizer	SGD; momentum 0.9; weight decay 1×10^{-4}
Learning rate	0.1, divided by 10 after epochs 30 and 60
Batch size; workers	4; 4 (Experiment 1: 2; 2)
Epochs	85; no early stopping
Checkpoint	Highest validation accuracy
Random seed	Not fixed

Table A2. Number of images per class in the training, validation, and test folders of the dataset (Experiment 3). Experiment 2 used the apple, background, banana, and tomato classes.

Class	Training	Validation	Test	Total
Apple	105	33	15	153
Background	106	43	21	170
Banana	106	32	15	153
Yellow pitahaya	110	35	20	165
Sweet granadilla	110	35	20	165
Papaya	110	35	20	165
Tree tomato	110	40	21	171
Tomato	107	30	16	153
Total	864	283	148	1295

Table A3. Five-fold stratified cross-validation on the 1147 training and validation images of Experiment 3: best epoch and maximum validation accuracy per fold. SD, sample standard deviation.

Fold	Best Epoch	Maximum Validation Accuracy (%)
1	64	100.00
2	58	99.13
3	61	100.00
4	82	100.00
5	73	100.00
Mean ± SD		99.83 ± 0.39

References

1. United Nations Development Programme. Inclusive Insurance and Risk Financing in Ecuador: Snapshot and Way Forward 2024. 2024. Available online: https://www.undp.org/sites/g/files/zskgke326/files/2024-08/15-ecuador.pdf (accessed on 3 September 2025).
2. World Bank. Ecuador: Growing Resilient for a Better Future. 2024. Available online: https://documents1.worldbank.org/curated/en/099050924131583105/pdf/P17797312232cd05b1881f1bffe33c0cfd2.pdf (accessed on 2 March 2026).
3. Ministerio de Producción, Comercio Exterior, Inversiones y Pesca. Informe de Gestión Institucional 2023. 2024. Available online: https://www.produccion.gob.ec/wp-content/uploads/2024/04/INFORME-DE-GESTION-MPCEIP-2023.pdf (accessed on 16 March 2026).
4. Food and Agriculture Organization of the United Nations. Banana Market Review 2022. 2023. Available online: https://openknowledge.fao.org/server/api/core/bitstreams/cd3e1df8-6e70-461a-9963-9827ad69389f/content (accessed on 16 March 2026).
5. Ministerio de Producción, Comercio Exterior, Inversiones y Pesca. Pitahaya Rompe Récord en Exportaciones con más de USD 172 Millones en 2023. 2023. Available online: https://www.produccion.gob.ec/pitahaya-rompe-record-en-exportaciones-con-mas-de-usd-172-millones-en-2023/ (accessed on 16 March 2026).
6. USDA Foreign Agricultural Service. Grain and Feed Annual. GAIN Report EC2025-0004. 2025. Available online: https://apps.fas.usda.gov/newgainapi/api/Report/DownloadReportByFileName?fileName=Grain+and+Feed+Annual_Quito_Ecuador_EC2025-0004.pdf (accessed on 16 March 2026).
7. World Intellectual Property Organization. Global Innovation Index 2022. 2022. Available online: https://www.wipo.int/global_innovation_index (accessed on 16 March 2026).
8. United Nations Conference on Trade and Development. Technology and Innovation Report 2023: Frontier Technologies Readiness Index. 2023. Available online: https://unctad.org/publication/technology-and-innovation-report-2023 (accessed on 16 March 2026).
9. Gaona-Gonzaga, P.; Vásquez-Rojas, L.; Aguayo-Pacas, S.; Viera-Arroyo, W.; Viteri-Díaz, P.; Sotomayor-Correa, A.; Medina-Rivera, L.; Mejía-Bonilla, P.; Cartagena-Ayala, Y. Respuesta del cultivo de granadilla (*Passiflora ligularis* Juss) cultivar "Colombiana" al suministro de nitrógeno y potasio por fertirriego. *Manglar* **2020**, *17*, 75–82.
10. Huachi, L.; Yugsi, E.; Paredes, M.F.; Coronel, D.; Verdugo, K.; Santamaría, P.C. Desarrollo de la pitahaya (*Cereus* sp.) en Ecuador. *Granja* **2015**, *22*, 50–58.

11. León F.,J.; Viteri D., P.; Cevallos A., G. *Manual del Cultivo de Tomate de Árbol*; Technical Report Manual No. 61; Instituto Nacional Autónomo de Investigaciones Agropecuarias (INIAP), Estación Experimental Santa Catalina, Programa de Fruticultura: Quito, Ecuador, 2004. Available online: https://repositorio.iniap.gob.ec/handle/41000/827 (accessed on 30 March 2026).
12. Vargas, Y.; Díaz, A.; Congo, C.; Tinoco, L.; Viera, W. Comparación de las características de calidad de fruta en genotipos de papaya (Carica papaya L.) provenientes de Shushufindi y La Joya de los Sachas, Ecuador. *Cienc. Tecnol. Agropecu.* **2021**, *22*, e1930. https://doi.org/10.21930/rcta.vol22_num1_art:1930.
13. Cubero, S.; Aleixos, N.; Moltó, E.; Gómez-Sanchis, J.; Blasco, J. Advances in Machine Vision Applications for Automatic Inspection and Quality Evaluation of Fruits and Vegetables. *Food Bioprocess Technol.* **2011**, *4*, 487–504. https://doi.org/10.1007/s11947-010-0411-8. Correction in *Food Bioprocess Technol.* **2011**, *4*, 829–830.
14. Gill, H.S.; Murugesan, G.; Mehbodniya, A.; Sajja, G.S.; Gupta, G.; Bhatt, A. Fruit Type Classification Using Deep Learning and Feature Fusion. *Comput. Electron. Agric.* **2023**, *211*, 107990. https://doi.org/10.1016/j.compag.2023.107990.
15. Khan, Z.; Rawat, S.; Kumar, S.; Sharma, A. Using Computer Vision for Automated Grading and Sorting of Fruits and Vegetables. *Int. J. Agric. Food Sci.* **2025**, *7*, 1056–1065. https://doi.org/10.33545/2664844X.2025.v7.i8j.674.
16. Koirala, A.; Walsh, K.B.; Wang, Z.; McCarthy, C. Deep Learning: Method Overview and Review of Use for Fruit Detection and Yield Estimation. *Comput. Electron. Agric.* **2019**, *162*, 219–234. https://doi.org/10.1016/j.compag.2019.04.017.
17. Kamilaris, A.; Prenafeta-Boldú, F.X. Deep learning in agriculture: A survey. *Comput. Electron. Agric.* **2018**, *147*, 70–90. https://doi.org/10.1016/j.compag.2018.02.016.
18. Mureşan, H.; Oltean, M. Fruit Recognition from Images Using Deep Learning. *Acta Univ. Sapientiae Inform.* **2018**, *10*, 26–42. https://doi.org/10.2478/ausi-2018-0002.
19. Duranton, M.; De Bosschere, K.; Cohen, A.; Maebe, J.; Munk, H. HiPEAC Vision 2015. 2015. Available online: https://www.hipeac.net/files/public/24/5/hipeac-vision-2015-Y5POWtDM.pdf (accessed on 16 March 2026).
20. Hartenstein, R. The Relevance of Reconfigurable Computing. In *Reconfigurable Computing: From FPGAs to Hardware/Software Codesign*; Cardoso, J.M.P.; Hübner, M., Eds.; Springer: New York, NY, USA, 2011; pp. 7–34. https://doi.org/10.1007/978-1-4614-0061-5_2.
21. NVIDIA. Edge Computing Solutions. 2026. Available online: https://www.nvidia.com/en-us/edge-computing/ (accessed on 16 March 2026).
22. Barlas, G. GPU Programming. In *Multicore and GPU Programming: An Integrated Approach*; Morgan Kaufmann: Waltham, MA, USA, 2015; pp. 391–526. https://doi.org/10.1016/B978-0-12-417137-4.00006-X.
23. Zahran, M. *Heterogeneous Computing: Hardware and Software Perspectives*; ACM: New York, NY, USA, 2019. https://doi.org/10.1145/3281649.
24. NVIDIA. NVIDIA TensorRT: High-Performance Deep Learning Inference Optimizer and Runtime. 2026. Available online: https://developer.nvidia.com/tensorrt (accessed on 16 March 2026).
25. Brodtkorb, A.R.; Dyken, C.; Hagen, T.R.; Hjelmervik, J.M.; Storaasli, O.O. State-of-the-art in heterogeneous computing. *Sci. Program.* **2010**, *18*, 540159. https://doi.org/10.3233/SPR-2009-0296.
26. Ramírez Gil, W.A.; Ramírez Gil, C.M. *Programación de Inteligencia Artificial: Curso Práctico*; RA-MA Editorial: Madrid, Spain, 2023. Available online: https://www.ra-ma.es/libro/programacion-de-inteligencia-artificial-curso-practico_147608/ (accessed on 30 March 2026).
27. Jing, E.; Zhang, H.; Li, Z.; Liu, Y.; Ji, Z.; Ganchev, I. ECG heartbeat classification based on an improved ResNet-18 model. *Comput. Math. Methods Med.* **2021**, *2021*, 6649970. https://doi.org/10.1155/2021/6649970.
28. Chanampe, H.; Aciar, S.; de la Vega, M.; Molinari Sotomayor, J.L.; Carrascosa, G.; Lorefice, A. Modelo de redes neuronales convolucionales profundas para la clasificación de lesiones en ecografías mamarias. In Proceedings of the XXI Workshop de Investigadores en Ciencias de la Computación (WICC 2019), San Juan, Argentina, 25–26 April 2019. Available online: http://sedici.unlp.edu.ar/handle/10915/77381 (accessed on 30 March 2026).
29. Möller, D.P.F. Machine Learning and Deep Learning. In *Guide to Cybersecurity in Digital Transformation: Trends, Methods, Technologies, Applications and Best Practices*; Möller, D.P., Ed.; Springer Nature: Cham, Switzerland, 2023; pp. 347–384. https://doi.org/10.1007/978-3-031-26845-8_8.
30. Alzubaidi, L.; Zhang, J.; Humaidi, A.J.; Al-Dujaili, A.; Duan, Y.; Al-Shamma, O.; Santamaría, J.; Fadhel, M.A.; Al-Amidie, M.; Farhan, L. Review of Deep Learning: Concepts, CNN Architectures, Challenges, Applications, Future Directions. *J. Big Data* **2021**, *8*, 53. https://doi.org/10.1186/s40537-021-00444-8.
31. NVIDIA. Jetson Nano Developer Kit User Guide. 2020. Available online: https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit (accessed on 5 October 2023).
32. NVIDIA. Jetson Nano. 2021. Available online: https://developer.nvidia.com/embedded/jetson-nano (accessed on 16 March 2026).
33. Mittal, S. A Survey on Optimized Implementation of Deep Learning Models on the NVIDIA Jetson Platform. *J. Syst. Archit.* **2019**, *97*, 428–442. https://doi.org/10.1016/j.sysarc.2019.01.011.

34. NVIDIA. JetPack Software Stack for NVIDIA Jetson. 2018. Available online: https://developer.nvidia.com/embedded/jetpack (accessed on 25 May 2025).
35. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*; IEEE: New York, NY, USA, 2016. https://doi.org/10.1109/CVPR.2016.90.
36. Pérez-Aguilar, D.A.; Risco-Ramos, R.H.; Casaverde-Pacherrez, L. Transfer learning en la clasificación binaria de imágenes térmicas. *Ingenius* **2021**, *26*, 71–86. https://doi.org/10.17163/ings.n26.2021.07.
37. Edel, G.; Kapustin, V. Exploring of the MobileNet V1 and MobileNet V2 Models on NVIDIA Jetson Nano Microcomputer. *J. Phys. Conf. Ser.* **2022**, *2291*, 012008. https://doi.org/10.1088/1742-6596/2291/1/012008.
38. Liu, W.; Anguelov, D.; Erhan, D.; Szegedy, C.; Reed, S.; Fu, C.Y.; Berg, A.C. SSD: Single Shot MultiBox Detector. In *Computer Vision—ECCV 2016*; Lecture Notes in Computer Science, Vol. 9905; Springer: Cham, Switzerland, 2016; pp. 21–37. https://doi.org/10.1007/978-3-319-46448-0_2.
39. Girshick, R. Fast R-CNN. In *2015 IEEE International Conference on Computer Vision (ICCV)*; IEEE: New York, NY, USA, 2015; pp. 1440–1448. https://doi.org/10.1109/ICCV.2015.169.
40. Redmon, J.; Divvala, S.; Girshick, R.; Farhadi, A. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*; IEEE: New York, NY, USA, 2016; pp. 779–788. https://doi.org/10.1109/CVPR.2016.91.
41. Zhang, Y.; Yu, J.; Chen, Y.; Yang, W.; Zhang, W.; He, Y. Real-Time Strawberry Detection Using Deep Neural Networks on Embedded System (Rtsd-Net): An Edge AI Application. *Comput. Electron. Agric.* **2022**, *192*, 106586. https://doi.org/10.1016/j.compag.2021.106586. Correction in *Comput. Electron. Agric.* **2023**, *215*, 108445.
42. Kumar, A.; Behera, L. High-Speed Detector for Low-Powered Devices in Aerial Grasping. *IEEE Robot. Autom. Lett.* **2024**, *9*, 4623–4630. https://doi.org/10.1109/lra.2024.3376997.
43. Ünal, H.B.; Vural, E.; Savaş, B.K.; Becerikli, Y. Fruit Recognition and Classification with Deep Learning Support on Embedded System (FruitNet). In *2020 Innovations in Intelligent Systems and Applications Conference (ASYU)*; IEEE: New York, NY, USA, 2020; pp. 1–5. https://doi.org/10.1109/ASYU50717.2020.9259881.
44. Ananthanarayana, T.; Ptucha, R.; Kelly, S.C. Deep Learning Based Fruit Freshness Classification and Detection with CMOS Image Sensors and Edge Processors. In *IS&T International Symposium on Electronic Imaging 2020: Food and Agricultural Imaging Systems*; Society for Imaging Science and Technology: Springfield, VA, USA, 2020; pp. 172-1–172-7. https://doi.org/10.2352/issn.2470-1173.2020.12.fais-172.
45. Singh, R.; Gill, S.S. Edge AI: A Survey. *Internet Things -Cyber-Phys. Syst.* **2023**, *3*, 71–92. https://doi.org/10.1016/j.iotcps.2023.02.004.
46. Sipola, T.; Alatalo, J.; Kokkonen, T.; Rantonen, M. Artificial Intelligence in the IoT Era: A Review of Edge AI Hardware and Software. In *2022 31st Conference of Open Innovations Association (FRUCT)*; IEEE: New York, NY, USA, 2022; pp. 320–331. https://doi.org/10.23919/FRUCT54823.2022.9770931.
47. Fauziah, F.; Yakti, B.K.; Prayitno, R.H.; Nurti, T.D.; Azizah, N. Classify Tomato Fruit Images Using Convolutional Neural Network (CNN) Method. *CCIT J.* **2024**, *17*, 59–69. https://doi.org/10.33050/ccit.v17i1.3072.
48. Huanca Namuche, A.A.; Terry Alvarado, B.S.; García Estrella, C. Redes neuronales convolucionales para la clasificación de la mancha negra en los cítricos. *Sapiens Int. Multidiscip. J.* **2025**, *2*, 1–20. https://doi.org/10.71068/4x9sw526.
49. Ji, W.; Pan, Y.; Xu, B.; Wang, J. A Real-Time Apple Targets Detection Method for Picking Robot Based on ShufflenetV2-YOLOX. *Agriculture* **2022**, *12*, 856. https://doi.org/10.3390/agriculture12060856.
50. Yang, Y.; Han, Y.; Li, S.; Li, H.; Zhang, M. Multi-Growth Period Tomato Fruit Detection Using Improved Yolov5. *Int. J. Robot. Autom. Technol.* **2022**, *9*, 44–55. https://doi.org/10.31875/2409-9694.2022.09.06.
51. Xu, Z.; Huang, X.; Huang, Y.; Sun, H.; Wan, F. A Real-Time Zanthoxylum Target Detection Method for an Intelligent Picking Robot under a Complex Background, Based on an Improved YOLOv5s Architecture. *Sensors* **2022**, *22*, 682. https://doi.org/10.3390/s22020682.
52. Li, X.; Wang, X.; Ong, P.; Yi, Z.; Ding, L.; Han, C. Fast Recognition and Counting Method of Dragon Fruit Flowers and Fruits Based on Video Stream. *Sensors* **2023**, *23*, 8444. https://doi.org/10.3390/s23208444.
53. Zhang, Y.D.; Dong, Z.; Chen, X.; Jia, W.; Du, S.; Muhammad, K.; Wang, S.H. Image Based Fruit Category Classification by 13-Layer Deep Convolutional Neural Network and Data Augmentation. *Multimed. Tools Appl.* **2019**, *78*, 3613–3632. https://doi.org/10.1007/s11042-017-5243-3.
54. Khan, R.; Debnath, R. Multi class fruit classification using efficient object detection and recognition techniques. *Int. J. Image, Graph. Signal Process.* **2019**, *11*, 1–18. https://doi.org/10.5815/ijigsp.2019.08.01.
55. Chaudhary, V.; Kumar, M.; Chaudhary, V.; Kumar, R.; Chauhan, C. Artificial Intelligence in Sustainable Horticultural Production: A Systematic Review. *Discov. Plants* **2025**, *2*, 372. https://doi.org/10.1007/s44372-025-00455-2.
56. Lipiński, S.; Sadkowski, S.; Chwietczuk, P. Application of AI in Date Fruit Detection—Performance Analysis of YOLO and Faster R-CNN Models. *Computation* **2025**, *13*, 149. https://doi.org/10.3390/computation13060149.

57. Nashrulloh, M.H.; Subarkah, A. Fruit Image Classification Using CNN With EfficientNet Architecture for Visual Education. *J. Intell. Syst. Technol. Inform.* **2025**, *1*, 67–71. https://doi.org/10.64878/jistics.v1i2.9.
58. Liu, X.; Zhang, Z.; Yu, J.; Hua, W.; Li, X.; Azizi, A.; Mhamed, M.; Abid, F.; Abdelhamid, M.A. Design, Integration, and Evaluation of a Low-Cost System for Automatic Apple Picking and Infield Sorting. *Comput. Electron. Agric.* **2025**, *239*, 110933. https://doi.org/10.1016/j.compag.2025.110933.
59. Tang, Y.; Chen, M.; Wang, C.; Luo, L.; Li, J.; Lian, G.; Zou, X. Recognition and Localization Methods for Vision-Based Fruit Picking Robots: A Review. *Front. Plant Sci.* **2020**, *11*, 510. https://doi.org/10.3389/fpls.2020.00510.
60. Lin, G.; Tang, Y.; Zou, X.; Wang, C. Three-Dimensional Reconstruction of Guava Fruits and Branches Using Instance Segmentation and Geometry Analysis. *Comput. Electron. Agric.* **2021**, *184*, 106107. https://doi.org/10.1016/j.compag.2021.106107.
61. Suntaxi Llumiquinga, A.M. Desarrollo de un Sistema de Clasificación de Frutas Nativas en Tiempo Real Utilizando GPU Embebido. Bachelor's Thesis, Universidad de las Fuerzas Armadas ESPE, Sangolquí, Ecuador, 2024. Available online: https://repositorio.espe.edu.ec/items/caee40a5-d291-43f2-a22e-68008407310b (accessed on 16 March 2026).
62. NVIDIA. Jetson Nano System-on-Module Data Sheet. 2024. Available online: https://developer.nvidia.com/embedded/downloads (accessed on 25 May 2025).
63. Franklin, D. jetson-inference: Hello AI World. 2025. Available online: https://github.com/dusty-nv/jetson-inference (accessed on 16 March 2026).
64. NVIDIA. *NVIDIA Jetson Software*, version JetPack 4.6.5 ; NVIDIA: Santa Clara, CA, USA, 2025. Available online: https://developer.nvidia.com/embedded/develop/software (accessed on 26 May 2025).
65. Prakosa, S.W.; Leu, J.-S.; Hsieh, H.-Y.; Avian, C.; Bai, C.-H.; Vítek, S. Implementing a Compression Technique on the Progressive Contextual Excitation Network for Smart Farming Applications. *Sensors* **2022**, *22*, 9717. https://doi.org/10.3390/s22249717.
66. Tobiasz, R.; Wilczynski, G.; Graszka, P.; Czechowski, N.; Luczak, S. Edge Devices Inference Performance Comparison. *J. Comput. Sci. Eng.* **2023**, *17*, 51–59. https://doi.org/10.5626/jcse.2023.17.2.51.
67. Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; Chen, L.C. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*; IEEE: New York, NY, USA, 2018; pp. 4510–4520. https://doi.org/10.1109/CVPR.2018.00474.
68. Liu, D.; Kong, H.; Luo, X.; Liu, W.; Subramaniam, R. A Unified Optimization Approach for CNN Model Inference on Integrated GPUs. In Proceedings of the 48th International Conference on Parallel Processing (ICPP), Kyoto, Japan, 5–8 August 2019. https://doi.org/10.1145/3337821.3337839.
69. Han, Y.; Yuan, Z.; Pu, Y.; Xue, C.; Song, S.; Sun, G.; Huang, G. Latency-aware Spatial-wise Dynamic Networks. *arXiv* **2022**, arXiv:2210.06223. https://doi.org/10.48550/arxiv.2210.06223.
70. Hijji, M.; Yar, H.; Ullah, F.U.M.; Alwakeel, M.M.; Harrabi, R.; Aradah, F.; Cheikh, F.A.; Muhammad, K.; Sajjad, M. FADS: An Intelligent Fatigue and Age Detection System for Smart Surveillance. *Mathematics* **2023**, *11*, 1174. https://doi.org/10.3390/math11051174.
71. Bi, Y.; Xue, B.; Zhang, M. GhostShiftAddNet: More Features from Energy-Efficient Operations. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*; IEEE: New York, NY, USA, 2021.
72. Hernández, D.; Cecilia, J.M.; Cano, J.-C.; Calafate, C.T. Flood Detection Using Real-Time Image Segmentation from Unmanned Aerial Vehicles on Edge-Computing Platform. *Remote Sens.* **2022**, *14*, 223. https://doi.org/10.3390/rs14010223.
73. Zagitov, A.; Chebotareva, E.; Toschev, A.; Magid, E. Comparative Analysis of Neural Network Models Performance on Low-Power Devices for a Real-Time Object Detection Task. *Comput. Opt.* **2024**, *48*, 242–252. https://doi.org/10.18287/2412-6179-co-1343.
74. Gao, M.; Song, P.; Wang, F.; Liu, J.; Mandelis, A.; Qi, D. A novel deep convolutional neural network based on ResNet-18 and transfer learning for detection of wood knot defects. *J. Sens.* **2021**, *2021*, 4428964. https://doi.org/10.1155/2021/4428964.
75. Misra, S.; Jeon, S.; Lee, S.; Managuli, R.; Jang, I.S.; Kim, C. Multi-Channel Transfer Learning of Chest X-ray Images for Screening of COVID-19. *Electronics* **2020**, *9*, 1388. https://doi.org/10.3390/electronics9091388.
76. Howard, A.G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv* **2017**, arXiv:1704.04861. https://doi.org/10.48550/arXiv.1704.04861.
77. Tan, M.; Le, Q.V. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In Proceedings of the 36th International Conference on Machine Learning, Long Beach, CA, USA, 9–15 June 2019; Volume 97, pp. 6105–6114. Available online: https://proceedings.mlr.press/v97/tan19a.html (accessed on 30 March 2026).
78. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*; Curran Associates, Inc.: Red Hook, NY, USA, 2019; pp. 8024–8035. https://doi.org/10.48550/arXiv.1912.01703.
79. Ahmad, M.A.; Suaib, N.M.; Ismail, A.W. RGBD depth sensor frame data streaming to handheld device using WebRTC-based framework. In *2024 IEEE International Symposium on Consumer Technology (ISCT)*; IEEE: New York, NY, USA, 2024. https://doi.org/10.1109/ISCT62336.2024.10791163.

80. Franklin, D. Jetson-Inference: WebRTC Server. 2023. Available online: https://github.com/dusty-nv/jetson-inference/blob/master/docs/aux-streaming.md (accessed on 16 March 2026).
81. Rescorla, E. *WebRTC Security Architecture*; RFC 8827; Internet Engineering Task Force: Wilmington, DE, USA, 2021. https://doi.org/10.17487/RFC8827.
82. Mohanty, S.P.; Hughes, D.P.; Salathé, M. Using Deep Learning for Image-Based Plant Disease Detection. *Front. Plant Sci.* **2016**, *7*, 1419. https://doi.org/10.3389/fpls.2016.01419.

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.