UNIVERSITY OF TRENTO

Department of Industrial Engineering

Doctoral School in Materials, Mechatronics and Systems Engineering
XXXVII *Cycle*

Techniques and Applications for Efficient Low-power TinyML

Supervisor:
Prof. Davide Brunelli

PhD Candidate:
Andrea Albanese

Copyright Notice

ApplePies 2023. Lecture Notes in Electrical Engineering, vol 1110. Springer, Cham.
https://doi.org/10.1007/978-3-031-48121-5_17

This thesis also contains material that is currently submitted for publication. The material has not yet been published, and this thesis provides a short adapted version of its content. The submitted work is the following:

- A. Albanese, Y. Wang, D. Brunelli and D. Boyle, "Is That Rain? Understanding Effects on Visual Odometry Performance for Autonomous UAVs and Efficient DNN-based Rain Classification at the Edge," 2024, in arXiv preprint arXiv:2407.12663.

This thesis also contains material that is currently submitted for a patent publication. The material has not been published yet, and this thesis provides a short adapted version of its content. The submitted patent is the following:

- A. Albanese, F. Barchi, D. Brunelli, N. Elia and D. Gotta, "Method and System for Controlling a Shipment," 2023.

Abstract

Tiny machine learning (tinyML) is becoming popular in Internet of Things (IoT) systems to add intelligence to end nodes with limited resources. Nowadays, there are tens of billions of connected devices that exchange data wirelessly, making dense IoT networks. However, the data involved in interconnected devices is increasing their memory footprint, making the transmission of raw data over low-power wide-area networks a challenging and expensive step. TinyML implements in-situ data processing, ensuring the efficiency and reliability of IoT systems without overloading communication channels.

Developing tinyML systems is complex because it involves the implementation of the traditional ML algorithm and then its optimization and compression to ensure successful deployment in resource-constrained devices. However, ML algorithms range from simple systems (e.g., 1-layer feed-forward neural networks), to the most complex ones, such as deep neural networks (DNNs) with tens of hidden layers and millions of parameters. The optimization of DNNs is an active research area as it presents many challenges to deploy them in IoT end-nodes efficiently.

This dissertation presents a general framework aiming at developing tinyML systems. It investigates different ML algorithms and embedded platforms to validate the correct operation of the proposed framework. Furthermore, it selects different use cases to motivate and demonstrate the effectiveness of the proposed solution for developing tinyML algorithms in IoT systems. The use cases consist of real-world applications providing actual techniques and methods to implement tinyML algorithms in constrained devices successfully. Furthermore, this thesis provides clear evidence of the benefits of tinyML considering energy efficiency, reliability, and maintenance.

Finally, it improves the capability of standard tinyML systems with on-device learning techniques. In this way, it is possible to obtain tinyML systems which follow the trend of the environment, learning new patterns and reducing maintenance operations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, artificial intelligence (AI), machine learning (ML), and deep learning (DL) have gained significant popularity in academia and industrial applications. They can achieve impressive performance by increasing the quality of service compared to traditional solutions. For instance, computer vision applications widely employ DL algorithms because of their outstanding performance in image classification, detection, and recognition. However, such algorithms are characterized by a high computational demand limiting their usage in powerful devices without memory and energy constraints. This poses a challenge when using DL in IoT systems where simple devices with low-power resources are connected.

Considering the enormous progress in embedded devices that follow Moore's law, it is now reasonable to integrate ML functionalities and ubiquitous intelligence in end devices with limited resources. Therefore, many researchers have moved their attention to this gap to make the end nodes of an IoT network intelligent and implement the tinyML paradigm. TinyML unleashes intelligence even in small, low-power, and low-cost devices with the benefits related to edge computing. On the other hand, implementing tinyML algorithms introduces different challenges such as model optimization and compression, computational capability, reliability, and maintenance.

The counterpart of edge computing is called "cloud computing" which permits the off-loading of the computational burden in remote servers. However, this approach is extremely inefficient as it

involves the transmission of raw data, thus increasing the traffic of big data in wireless networks. Moreover, the wireless transmission of raw data involves a considerable latency and energy consumption which affects the frame rate and energy sustainability of any application. Lastly, it is a costly solution as it involves the usage of high-cost servers with near-unlimited resources [3, 4]. On the other side, cloud computing can execute the most complex ML algorithms, thus guaranteeing a high quality of service [5].
TinyML with the edge computing approach fits perfectly with IoT systems and overcomes all the limitations of cloud computing permitting the deployment of ML algorithms close to the data source (e.g., the sensor itself).
The main benefits of tinyML are:

- **Near-sensor computing:** Raw data are processed as close as possible to the data source avoiding the transmission of raw data. In this way, it is possible to send only the processed data with the useful information, thus saving transmission energy and data storage. The most recent and advanced sensors (e.g., image sensors with in-pixel processing capabilities) permit the processing of the acquired data directly in the acquisition pipeline with simple but effective ML solutions [6]. This reduces the energy demand and the runtime execution by keeping a high quality of service.

- **Real-time execution:** TinyML reduces the communication latency by moving the inference to the source device. This guarantees a real-time execution even in time-sensitive applications where a prompt exchange of information is fundamental. In particular, mobile robotics applications require real-time responsiveness making the tinyML a successful approach for such scenarios.

- **Privacy concerns:** TinyML avoids the transmission of raw data, thus it ensures the privacy of every individual without sending over the internet any sensitive data. It limits the usage of sensitive data for in-situ processing and then deletes them. Thus, it protects data privacy from any cyber attack.

- **Network independent:** TinyML processes data locally, thus it does not indispensably need an always-available network for wireless data transmission. This opens the usage of tinyML

in extreme scenarios where a wireless network might not be available.

- **Cost-effective:** TinyML devices are low-cost permitting large-scale manufacturing and small, thus deployable in hard-to-reach positions. Such devices range from embedded GPUs, and single-board computers, to microcontrollers with a few kilobytes of memory.

To take advantage of all these benefits, it is fundamental to develop ML models aware of the hardware used. This introduces the main challenge of tinyML: deploying complex ML models, such as DNNs, on end devices with low computational capabilities. Fortunately, there are existing solutions for model compression and optimization that permit the deployment of complex ML models in resource-constrained devices. However, compression and optimization techniques introduce performance degradation, thus it is important to optimize models by keeping an acceptable quality of service. For this reason, researchers are working in this field to find solutions that can highly compress models without significantly affecting performance.
The main effective optimization techniques are:

- **Quantization:** It converts model weights represented in floating point (i.e., 32-bit), with an integer representation (i.e., 8-bit). This introduces a 4× compression factor. However, it is important to test the quantized model as it could not satisfy the application requirements due to the lack of bit precision.

- **Pruning:** It is used in deep architectures where weights and branches are uninfluent (i.e., close to 0), or rarely activated. Then, it cuts off those branches that do not play an active role in producing a fair result, obtaining a parameter reduction. After the pruning operation, it is needed a fine-tuning of the new architecture.

- **Parameter optimization:** ML models, especially DNNs, are characterized by a high number of parameters. However, depending on the application scenario, a lower number of parameters can satisfy the system requirements. Thus, it is possible to simplify the model architecture by removing layers and reducing the overall number of parameters. This operation is

performed empirically to find the best trade-off between memory footprint and performance.

- **Knowledge distillation:** This technique trains a simple network (i.e., a student architecture) starting from a complex architecture, namely the teacher architecture. This achieves a lightweight model with a similar performance to the teacher model.

The tinyML paradigm presented above presents a main limitation: such systems use static models that can only perform inference, making them vulnerable to context drift (i.e., the effect related to the changing of the environment). This leads to performance degradation and periodic maintenance due to model retraining and fine-tuning. Thus, ensuring a high model accuracy over time requires continuous monitoring and potential retraining, which is harder on distributed, resource-constrained devices. This challenge is solved with the so-called 'on-device training' or 'online learning'. It performs inference and training on the input data received, thus learning from the changes that affect the environment and recognising new patterns during runtime. However, ML model training, especially for computer vision applications, is typically performed on desktop GPUs because it requires a high computational capability, thus moving the training step on resource-constrained end devices is almost impossible. With 'tiny online learning' (tinyOL) techniques, it is possible to add learning capabilities to end devices without affecting the application latency considerably. It uses trade-offs such as the combination of a static model with a dynamic model which can be updated during runtime. This increases the lifetime of IoT devices reducing the maintenance and increasing the performance over time.
TinyML is widely employed in academia, however, many real-world applications do not benefit from it because of the development complexity. Even though libraries such as TensorFlow Lite and PyTorch Mobile support tinyML development, the ecosystem is still fragmented, making its implementation difficult. Thus, the lack of standardized tools and frameworks makes the tinyML integration into real-world applications challenging. Moreover, many industries are not adopting tinyML due to the lack of robustness and integration with existing systems. In general, the insufficiency of tinyML widespread and high-profile success case studies makes companies

hesitant to invest in tinyML systems [7, 8].
This dissertation aims to provide a framework and methodologies to develop efficient tinyML systems to overcome the challenges related to development complexity and maintenance. This removes the gap between tinyML research and its employment for commercial purposes opening its usage in many fields. The effectiveness of the proposed framework and methodologies is proven with real-life applications of tinyML and tinyOL which give clear evidence of their benefits in various scenarios. Moreover, they demonstrate the improvements in moving the computational burden near the sensor compared to cloud solutions. Finally, they address the analysis, comparison, and characterisation of different hardware classes to find the optimal trade-off between computational demand, memory usage, performance, and energy efficiency. In particular, the following main use cases are explored:

- **Precision agriculture:** Agriculture fields have limited resources requiring low-impact monitoring systems. TinyML can give precious support in this scenario by integrating advanced monitoring for pests and weed infestation, thus optimising the usage of pesticides.

- **Autonomous drones:** Mobile robotics, in particular drones, are increasingly autonomous thanks to advanced sensors and navigation techniques. However, small-sized drones have limited payload and energy resources which make fundamental the usage of tinyML for implementing autnomous navigation with advanced data processing.

- **Industrial inspection:** Companies are changing their inspection systems from expensive devices to low-power and low-cost hardware. In this scenario, tinyML and tinyOL support the integration of efficient monitoring systems into low-cost devices with learning capabilities.

The thesis is organized into 7 chapters. **Chapter 1** introduces the thesis by giving an overview of the main benefits and challenges of tinyML.
Chapter 2 presents an innovative smart trap with tinyML functionalities. It first introduces the scenario highlighting the main challenges. Then, it presents the implementation of the pest detection system based on deep neural networks running on a single-

board computer. Finally, it presents the platform characterization ensuring its energy neutrality thanks to the usage of a solar energy harvester.
Chapter 3 addresses the autonomous navigation of small-size drones with an unintrusive and low-power automatic landing system. Furthermore, it analyses the disturbances introduced by adverse weather conditions, in particular rain, and provides a tinyML-based solution to predict and counteract critical conditions.
Chapter 4 includes the implementation of an industrial visual inspection system based on MCU-based cameras and deep neural networks running efficiently on the edge. The results confirm the benefits of using tinyML in this use case.
In **Chapter 5**, the tinyOL system is presented by focusing on the main challenges and possible solutions to execute such algorithms in MCUs. Furthermore, it presents a further extension which involves unsupervised learning techniques with tinyOL to extend its usage in real-world applications.
Chapter 6 concludes the thesis with final remarks and proposes future work to push tinyML systems at the extreme edge.
Finally, **Chapter 7** presents the main activities carried out during this research experience.

Chapter 2

Precision Agriculture with TinyML

Artificial intelligence has smoothly penetrated several economic activities, especially monitoring and control applications, including the agriculture sector. However, research efforts toward low-power sensing devices with fully functional machine learning (ML) onboard are still fragmented and limited in smart farming. Biotic stress is one of the primary causes of crop yield reduction. With the development of deep learning in computer vision technology, autonomous detection of pest infestation through images has become an important research direction for timely crop disease diagnosis. This chapter [1] presents an embedded system enhanced with ML functionalities, ensuring the detection of pest infestation inside fruit orchards. The embedded solution is based on a low-power embedded sensing system along with a neural accelerator able to capture and process images inside common pheromone-based traps. Three dif-

[1] The work presented in this chapter has been published in the following papers:

- Brunelli, D., Albanese, A., d'Acunto, D., and Nardello, M. (2019). Energy neutral machine learning based iot device for pest detection in precision agriculture. IEEE Internet of Things Magazine, 2(4), 10-13.
- Albanese, A., Nardello, M., and Brunelli, D. (2021). Automated pest detection with DNN on the edge for precision agriculture. IEEE Journal on Emerging and Selected Topics in Circuits and Systems, 11(3), 458-467.

ferent ML algorithms have been trained and deployed, highlighting the capabilities of the platform. Moreover, the proposed approach guarantees an extended battery life thanks to the integration of energy harvesting functionalities. Results show how it is possible to automate the task of pest infestation for unlimited time without the farmer's intervention.

2.1 Introduction

Due to the constant growth of food production demand, in Europe, agriculture is responsible for more than 10% of greenhouse gas emissions and 44% of water consumption nowadays. Chemical treatments (e.g., pesticides) are being intensively used to improve the market penetration of fruit crops, leading to a remarkable impact on pollinators and the planet's ecosystem. Thus, there is an increasing interest in new techniques to lower the water demand [9] and optimize pesticide treatments to preserve natural resources[2].
Farmers and researchers have been teaming up to develop smart systems for precision agriculture. Networks of smart sensors are mounted directly inside fruit and vegetable orchards, and advanced machine learning algorithms optimize the agriculture resources usage, enabling crop monitoring by gathering real-time data about the orchard's health.
Usually, pheromone-based traps are equipped with a passive camera that captures pictures of the trapped insects and sends them through internet nodes to the cloud. Afterward, an expert, or a farmer, is required to review the captured images to check the effective presence of dangerous parasites and eventually plan a local counteraction (i.e., a pesticide treatment). However, this process requires a high amount of data to be sent over long-range communication, making the application inefficient and time-expensive due to regular human intervention in the detection process.
Recently, researchers have started investigating smart-trap usage for pest detection as a solution to increase the wealth of orchards while lowering pesticide demand [10]. These traps – installed directly inside orchards – can autonomously detect dangerous parasites and alert the farmer to apply targeted pesticide treatments. Thanks to

[2]Source: `https://projects.research-and-innovation.ec.europa.eu/en/horizon-magazine/how-crop-and-animal-sensors-are-making-farming-smarter`

the implementation of sophisticated at-the-edge machine learning algorithms using the tinyML paradigm, the smart trap can detect dangerous parasites without remote cloud infrastructure as generally required for machine learning applications.
This solution opens many new possibilities for monitoring applications in precision agriculture: i) The optimized usage of the limited energy available inside orchards; ii) The balanced distribution of the whole implemented application by exploiting the computation capabilities at different levels (edge-concentrators-cloud) for achieving better scalability; iii) The capability of using the recent low-power long-range and low-data-rate radio protocols [11, 12], as we do not need to send the acquired massive image data but only the result of its analysis; iv) Finally, thanks to a low energy budget, the capabilities of exploiting energy harvesting extending to unlimited lifetime the smart traps.
This chapter presents a smart trap for pest detection running a Deep Neural Network (DNN) on edge. The smart trap enables fast detection of pests in apple orchards by using ML algorithms that improve the overall system efficiency [13, 14]. All the computation is done on the node, thus slimming down the amount of data transmission and limiting it to a simple notification of a few bytes if threats are detected. The smart trap features an energy harvesting system composed of a real-time clock (RTC) to trigger the pest detection task – implemented using a low-power STM32L0[3] MCU –, a small Li-Ion battery, and a solar panel to power its operations indefinitely. The hardware solution is developed on top of a Raspberry Pi single-board-computer as a mainboard equipped with a camera as an image sensor and an Intel Neural Compute Stick (NCS) as a neural accelerator to optimize the inference execution and, accordingly, the energy consumption [15, 16, 17]. The overall system has been characterized by considering its power consumption over a full application cycle to find out the power-hungry tasks that require dedicated power optimization to meet the system energy balance. Furthermore, three different CNN models for image classification have been compared: a modified LeNet-5 [18] which, thanks to its straightforward architecture, can speed the execution up, a VGG16 [19, 20] model which features a more deep CNN architecture capable of achieving better recognition accuracy when classifying class of objects, and finally a

[3]Info: `https://www.st.com/en/microcontrollers-microprocessors/stm32l0-series.html`

MobileNetV2 [2] network perfectly suitable to be evaluated on the edge by resource-constrained platforms.
This chapter explores the following aspects:

- The development and characterization of IoT smart traps that can be deployed and left working unattended for prolonged times without the need for any human intervention;
- The study, training, optimization, and validation of three different ML models for image classification suitable for the resource-constrained embedded platform. Results and performance comparison are presented and determine the model used for the final deployment.
- The platform sustainability assessment when powered using the solar energy harvester.

2.2 Related Works

The recent advances in smart agriculture are mainly powered by the progress in wireless sensor networks (WSNs), unmanned autonomous vehicles (UAVs) [21], machine learning, low-power imaging systems [22], and cloud computing. Tiny sensors are deployed in difficult-to-access areas for the periodic acquisition of in-field data [23]. Although research and comparable prototypes are still limited, we provide a discussion on successful automating pest detection tasks.
Monitoring is a crucial component in pheromone-based pest control systems [24, 25]. In widely used trap-based pest monitoring, captured digital images are analyzed by human experts for recognizing and counting pests. Manual counting is labour-intensive, slow, expensive, and error-prone, which precludes real-time performance and cost targets.
Deep learning techniques are used to overcome these limitations and achieve a completely automated, real-time pest monitoring system by removing the human from the loop [26]. [27] is one of the recent works that exploit ML techniques to classify insects. These works can be grouped based on the applied method. In terms of image sources, many articles have investigated insect specimens [28, 29, 30]. Unfortunately, even if specimens are usually well preserved and managed in a laboratory environment, this approach is

not suitable for creating a model for image classification when data is collected in the wild. Researchers have thus proposed to extend the datasets with images captured from real traps [27, 31, 32].
From an algorithmic perspective, various types of features have been used for insect classification, including wing structures [28, 29, 30], morphometric measurement [33] and global image features [34, 35, 36, 37].
Different classifiers were also developed starting from the various feature extraction methods, including but not limited to support vector machines (SVM) [33, 38, 39], artificial neural networks (ANN) [40, 41, 42, 43] and k-nearest neighbours (KNN) [44, 45]. In general, these proposed methods were not tested under real application scenarios, for example, to classify real-time images acquired from real traps deployed inside orchards.
To solve some of these early methods' problems, more recently, deep learning has been proposed in the literature [46, 47, 48]. Their variants have emerged as the most effective method for object recognition and detection by achieving state-of-the-art performance on many well-recognized datasets also in the domain of precision agriculture. A particular class of DL algorithms – well known as Convolutional Neural Networks (CNN) – has made a clear breakthrough in computer vision techniques for pest detection. Many sorts of algorithms based on CNN have emerged, significantly improving current systems' performance for classification as well as object localization [49, 50, 51, 52, 53].
Inspired by this research line, we adopt a region-based detection pipeline with convolutional neural networks as the image classifier to classify in-situ images captured inside pheromone-based traps deployed inside apple orchards. The raw images are firstly preprocessed with colour correction. Then, trained ConvNets are applied to densely sampled image patches to extract single regions of interest (ROIs). ROIs are then filtered by non-maximum suppression, after which only those with probabilities higher than their neighbours are preserved. Finally, the remaining ROIs are thresholded. ROIs that meet the threshold are considered correct detections.
Even though the approach presented in this chapter is not a novelty, the proposed solution can provide state-of-the-art detection results directly from apple orchards. Our system shows a suitable accuracy for the implemented task without first uploading acquired data to the cloud. All the computation is carried out autonomously on the

edge, with the capability to exploit energy harvesting to avoid the energy overhead of the metering infrastructure and extend the lifetime of the installation.

2.3 System Architecture

The system has been designed to bring IoT technologies into a domain that requires data collection over vast areas. In this scenario, the system requires long-range communication, and high scalability from multiple devices. Thanks to the onboard recognition algorithm, the smart trap's data transmission is limited to a few bytes, thus exploitable in any rural area. Only the result of the inference will be transmitted, making it suitable even for low-bitrate communication. If the farmer needs a visual confirmation from the captured picture, a few images per day can be transmitted. Figure 2.1 presents the system prototype.

Figure 2.1: An example of the assembled prototype used during indoor testing.

2.3.1 Hardware Implementation

Each smart trap is built on a custom hardware platform that includes: a small, low-power image sensor to collect images; a compact single-board computer from Raspberry PI; an Intel Neural Compute module for optimizing the execution of the ML task; a long-range radio chip for communication; and a solar energy-harvesting power system for collecting and storing energy from the environment. Figure 2.2 presents the schematic block overview of the proposed IoT device. Its main functionalities are designed as follows.

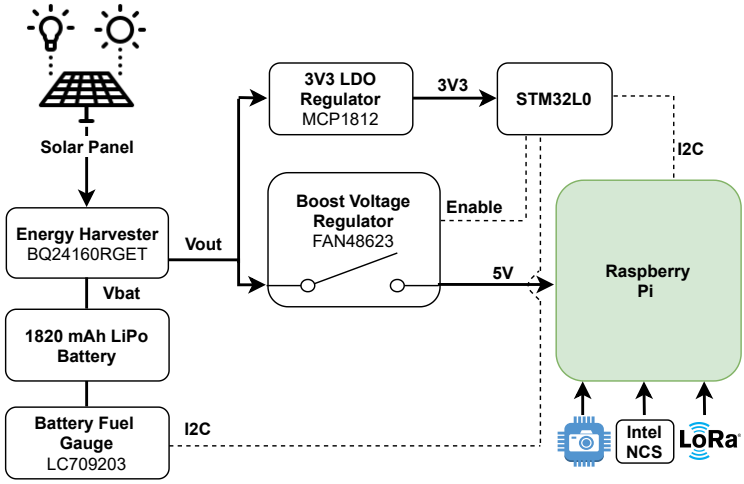


Figure 2.2: Solar energy harvester and power management circuit schematic block.

Sensing

The smart trap uses a Sony IMX219 image sensor. The IMX219 is a low-power back-illuminated CMOS image sensor. The sensor utilizes 1.12 µm pixel technology that offers high sensitivity and only needs a few external passive components to work. It integrates a black-levels calibration circuit, automatic exposure, and gain control loop to reduce host computation and commands to the sensor to optimize the system power consumption.

Processing

The processing layer is mainly composed of two parts. The first, the Raspberry single board computer is responsible for managing the sensor acquisition, the processing of the captured pictures, and the transmission. After several tests with different releases on the market, we select the Pi3 version as the best trade-off between computing capability, energy demand, and cost. The second part consists of a neural accelerator, namely an Intel Neural Compute Stick, that is activated only during the ML task and reduces the inference time.

Transmission

The smart trap is equipped with a LoRa module [54]. The connectivity is provided by an RFM95W transceiver featuring a LoRa low-power modem and a +20 dBm power amplifier that can achieve a sensitivity of over -148 dBm. The LoRa IC is then connected to an external antenna with a maximum gain of 2 dBi.

Power unit

The smart trap integrates a complete power supply with energy harvesting functionalities to efficiently use the LiPo battery's limited energy. Figure 2.2 presents the schematic block of the power supply module. Starting from the top left corner, the solar panel is connected directly to the energy harvester BQ24160, used to recharge an 1820mAh Li-Po battery. Two voltage converters are connected to it: the first, a MCP1812[4] LDO, is in charge of generating 3.3V to the external microcontroller. The second, a FAN48623[5] Boost converter, provides a stabilized 5V to the Raspberry. This converter is controlled by an STM32L0 MCU and enabled according to the implemented power policy (e.g., SW programmed intervals). A battery fuel gauge LC709203F[6] is used by the MCU for battery status monitoring. The external low-power STM32L0 MCU is in charge of the energy management of the platform. It enables the power-up and

[4]Info: `https://www.microchip.com/wwwproducts/en/MCP1812`

[5]Info: `https://www.onsemi.com/pdf/datasheet/fan48623-d.pdf`

[6]Info: `https://www.onsemi.com/download/data-sheet/pdf/lc709203f-d.pdf`

shutdown of the Raspberry and manages the harvesting functionalities, guaranteeing the optimal battery life without the farmer's intervention.

2.3.2 Detection pipeline

The smart trap implements a multi-stage pipeline that processes an image after its capture to detect dangerous insects, called Codling Moth. The automatic detection pipeline is shown in Figure 2.3.

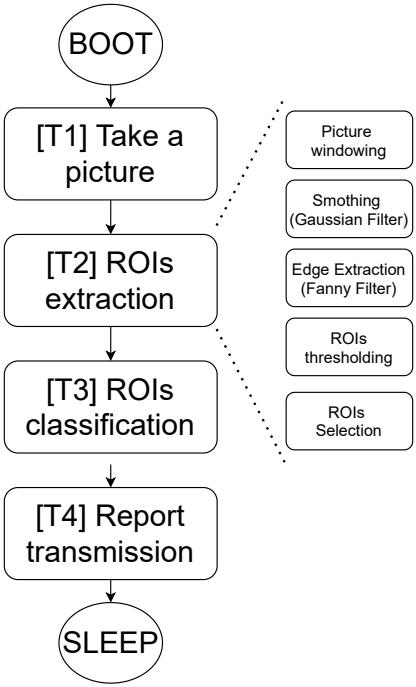
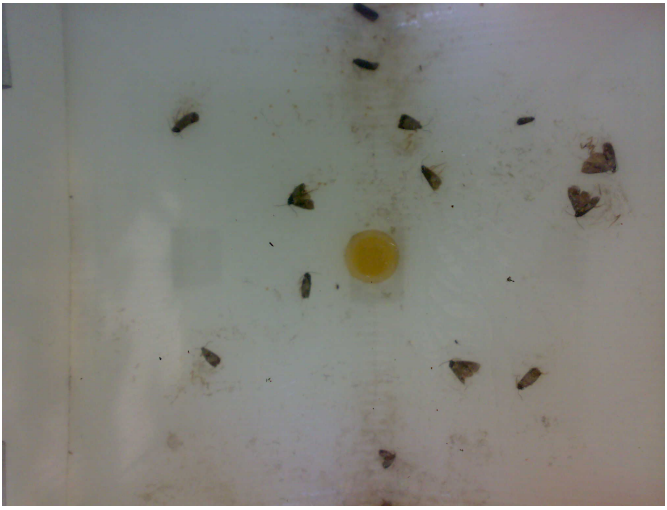


Figure 2.3: Detection Pipeline workflow. On the right, the ROI extraction procedure is highlighted.

We use sliding windows and a trained image classifier. The classifier is applied to local windows at different locations of the entire image to extract and classify the regions of interest (ROIs). ROIs are a portion of the captured image encompassing just a single insect. An example of this operation is depicted in Figure 2.4.



(a) Raw picture.

(b) RoI with codling moth.

(c) RoI with general insect.

Figure 2.4: Image (a) presents a photo taken inside a pheromone-based trap by the proposed smart camera while (b) and (c) present an example of extracted ROIs with a single insect during the pre-processing phase.

The classifier's output is a single scalar, representing the probability that a particular ROI contains a codling moth. These ROIs are regularly and densely arranged over the image and thus largely overlapping. Therefore, we perform smoothing (or blurring) of the frame with a Gaussian filter and then Edge extraction through the Canny operator to select only the ROIs whose respective probability

is locally maximal.
After this detection phase, ROIs are further analyzed by the learning algorithm that tries to assert if the detected insect is a codling moth or not. The final output of the detection pipeline is the initial captured images along with the coloured square highlighting the detected positive ROIs, as shown in Figure 2.5. After the DNN assessment, a report is generated and sent using the LoRa modem to the farmer.

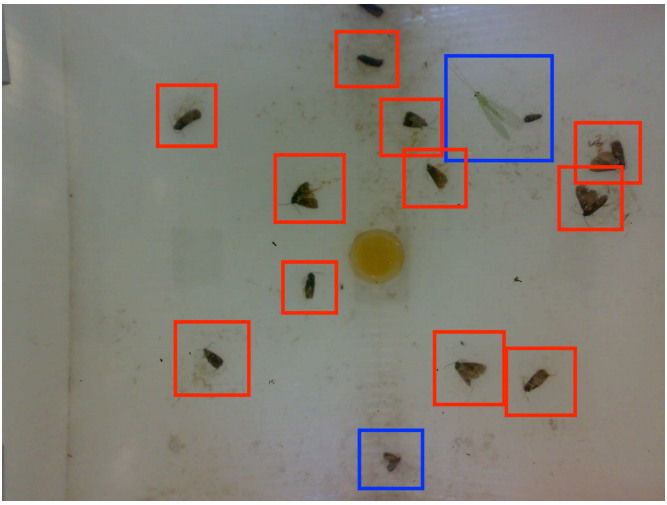


Figure 2.5: An example of an annotated photo after its evaluation. The red boxes highlight the detected codling moth (positive class) while the blue box general insects (negative class).

2.3.3 Edge Accelerator

Edge accelerator is a class of purpose-built Systems on a Chip (SoC) for running deep learning models efficiently on edge devices. Different companies have proposed hardware solutions to accelerate the execution of deep learning algorithms at the edge of the network. To this end, we took a systematic look at a set of edge accelerators, their working principles, and their performance in terms of executing different learning tasks. We compared three different platforms: Intel NCS2, Google Coral USB TPU and Nvidia Jetson Nano, which

are state of the art on the market.

Energy consumption. Energy is a precious resource in battery-powered edge accelerators. From an energy consumption viewpoint, the most power-hungry platform is the Jetson Nano, requiring up to 10W when exploiting the GPU during the inference[7]. On the contrary, the power consumption of the Google TPU is around 5W. A similar power consumption is measured for the Intel NCS 2 along with the carrier board (in our case, an RPi 3B+) [55], mainly divided into 2W consumed by the accelerator and 3W by the RPi.

Performance. The execution time of the model inference is a key metric for sensory systems. Among the 3 platforms compared, the Nvidia Jetson presents the higher computational capabilities, followed by the Google TPU [56]. The Intel NCS 2 is the less powerful platform, but still perfectly suitable for the proposed implementation that does not require hard real-time execution of the ML task. In our case, we are more focused on energy reduction; thus we selected the Intel NCS2 as a neural accelerator for the proposed application.

Compatibility. Although Edge TPU appears the most competitive in terms of performance and size, it is also the most limiting in software as only Tensorflow frameworks are supported. Moreover, it does not support the full Tensorflow Lite but only the models that are quantized to 8-bit integers (INT8). This contrasts with NCS2 which also supports FP16/32 (16/32-bit floating point) in addition to INT8. In addition, the Intel NCS2 is widely supported by the community, thanks to the OpenVINO[8] toolkit that allows the conversion of machine learning frameworks. Nvidia's software is the most versatile as its TensorRT supports most ML frameworks including MATLAB. Moreover, Google TPU and NCS2 are designed to support some subset of computational layers (primarily for computer vision tasks).

Availability. Hardware availability was also a factor limiting the platforms and configuration tested. At the time of our project kick-off, only the Intel NCS2 and the Google TPU were available and adequately documented.

The exploration of the design space suggested continuing our devel-

[7] Info: `https://docs.nvidia.com/jetson/archives/r34.1/DeveloperGuide/index.html`

[8] Info: `https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html`

opment on the Intel NCS2, because it represents the best trade-off in terms of performance, energy consumption and learning model compatibility.

2.4 Deep Neural Networks

Deep Learning is a class of machine learning algorithms based on the so-called ANNs trained through feature learning techniques. DL algorithms can improve the recognition capability of many systems by simulating the biological neural networks (i.e., the human brain behaviour). They can automatically learn features at multiple levels of abstraction and compose them to learn complex ones. For this application purpose, three state-of-the-art DNN architectures represented in Figures 2.6, 2.7 and 2.8 has been chosen:

- A modified **LeNet-5** [18], presented in Figure 2.6, which features a simple and straightforward structure. It has been designed for hand-written character recognition, but we extended it for classification problems with a few modifications. As revealed in Figure 2.6, it is composed of seven layers: 3 convolutional, 2 subsampling and one fully connected layer followed by a softmax classifier. Moreover, the second convolutional block does not use all the features produced by the average pooling layer. This permits the convolutional kernels to learn different patterns and improve classification accuracy. It also makes the network less computing demanding, which is suitable for embedded platforms. By changing the original activation function (i.e., *tanh*) to the rectified linear unit, it is possible to extend this network for classification tasks where specific patterns have to be recognized (especially for our case of pest detection).

- **VGG16** [19, 20], represented in Figure 2.7, is characterized by a complex architecture and, when compared with LeNet performance, reveals the higher potential in classification accuracy. It uses convolutional kernels 3×3 with stride 1 and the same padding and max-pool layers of 2×2 filter and stride 2. In this way, convolutional kernels can learn different patterns and geometrical shapes. Fully connected layers enable the classification of objects depending on the position of shapes

in the image. Thus, this network is perfect for recognition problems.

- **MobileNetV2** [2], shown in Figure 2.8, is based on an inverted residual structure where the shortcut connections are between the thin bottleneck layers. In MobileNetV2, there are two types of inverted residual blocks. The first is with a stride of 1 and the second with a stride of 2 for downsizing. The network is composed of 3 layers for both types of blocks. The first layer is 1×1 convolution with ReLU6. The second layer is the depthwise convolution. The third layer is another 1×1 convolution without any non-linearity. The intuition is that the bottlenecks encode the model's intermediate inputs and outputs while the inner layer encapsulates the model's ability to transform from lower-level concepts such as pixels to higher-level descriptors such as image categories. Thanks to this architecture, the network is perfectly suitable for building highly efficient mobile models. Finally, as with traditional residual connections, shortcuts enable faster training and better accuracy.

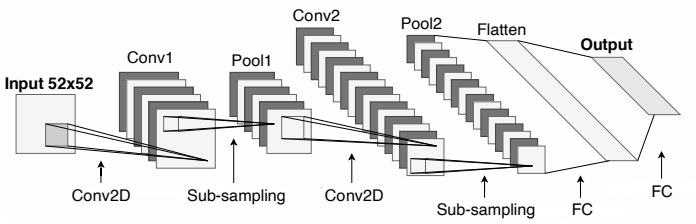


Figure 2.6: LeNet-5 architecture.

We tested these three learning model architectures to select the best trade-off between complexity, accuracy, and power consumption.

2.4.1 Training session

The dataset for the training phase has been constructed in a semi-automatic way thanks to a specific image processing algorithm. It starts from raw pictures, extracts relevant features such as contours and blobs, and generates tiles with codling moths or general insects

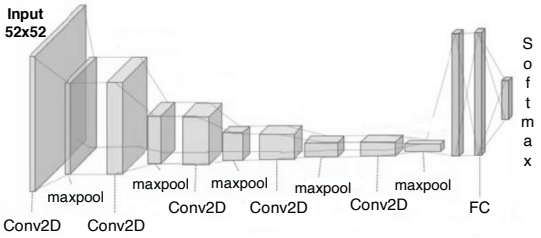


Figure 2.7: VGG16 architecture.

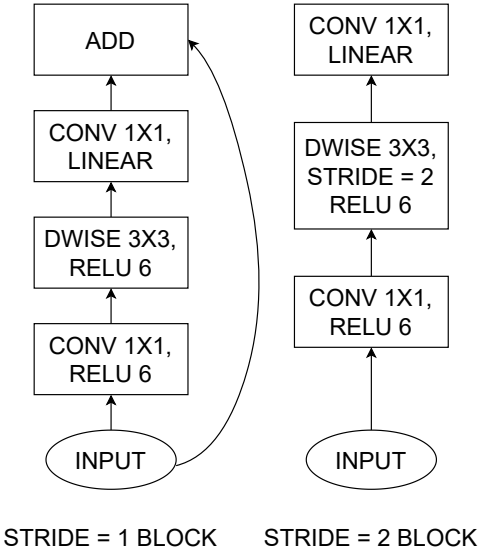
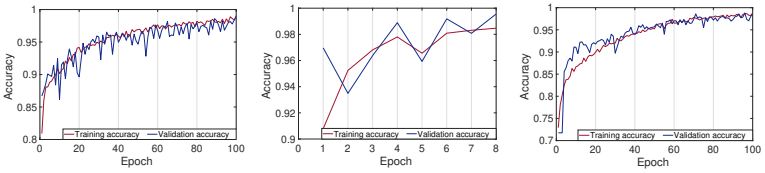


Figure 2.8: MobileNetV2 architecture.

for the dataset. Initially, the overall dataset consists of 1100 images where 30% are used for validation.
The number of dataset images has been increased by combining augmentation techniques such as dataset generation (before training) and in-place augmentation (during training). These techniques artificially expand the dimension of a training dataset and improve the dataset sparsity to prone the network to generalization capabilities. The purpose of these approaches is to create an expanded version of

the original dataset by applying various image processing operators (e.g., shift, flip, zoom). Finally, the overall dataset consists of 4400 images divided into two classes: 3200 for *codling moth* and 1200 for *general insect*. It has been further split into 3500 images for the train (2500 for *codling moth* and 1000 for *general insect*) and 900 for the test (625 for *codling moth* and 275 for *general insect*). An example of the dataset images is shown in Figure 2.4, depicting a raw picture used for the dataset construction (Figure 2.4a), a tile with Codling Moth (Figure 2.4b), and a tile with a general insect (Figure 2.4c) used for the network training.
The training is performed over 100 epochs with an input image size equal to 52×52 for LeNet-5, VGG16 and MobileNetV2. An early stopping by accuracy approach has been further used to stop the training if the validation accuracy reaches at least 99.5%. In this case, only VGG16 has stopped earlier than others (i.e., 8 epochs have been enough) as shown in Figure 2.9b.



(a) (b) (c)

Figure 2.9: Training and test accuracy comparison for (a) LeNet, (b) VGG16 and (c) MobileNetV2 .

Figure 2.9 shows the accuracy and validation for LeNet, VGG16 and MobileNetV2 over the epochs. Notice that all architectures' accuracy increases together with the validation accuracy and converges to almost 0.99. This confirms that the training sessions have been successful without overfitting. Moreover, VGG16 has reached the desired validation accuracy earlier than others because of its high number of parameters and its deep structure.

2.4.2 Validation

The obtained DNN is tested with images that have been retrieved from the dataset before the training phase. In this way, it is possible to test the network generalization capability with new images. As

stated before, 900 images have been used for the tests. The results for the three solutions are shown in Table 2.1. Notice that VGG16

Table 2.1: LeNet, VGG16 and MobileNetV2 test results.

(%)	**Accuracy**	**Recall**	**Precision**	**F-score**
LeNet-5	96.1	94.9	99.6	97.2
VGG16	97.9	97.4	99.6	98.5
MobileNetV2	95.1	94.5	98.5	96.4

features a high precision but a lower recall, which means that it misses some pests, but the predicted ones are accurate, and, consequently, the F-measure is good.
To evaluate if the metrics satisfy the application requirements, we interviewed apple farmers and looked into the literature. Commonly, farmers detect Codling Moth infestation exploiting pheromone-based traps. The traps are then checked once every week, and the pesticide treatment is executed when 2 moths/week [57] are trapped/detected. Considering the average number of Codling Moth caught seasonally, as reported in several articles [57, 58], the accuracy of the system is perfectly suitable for automating the Codling Moth detection task. Moreover, the smart trap is programmed to check for Codling Moth twice per day, allowing a prompter reaction compared to today's human-inspection approach. All architectures confirm their generalization capability if new images are given as input. Thus, it implies that the models have been trained without overfitting. On the other hand, both LeNet and MobileNetV2 present good results in all parameters; thus, the three architectures can be used in this application scenario. However, considering the deep architecture and its high number of parameters, VGG16 and MobileNet could outperform LeNet if the training dataset is even more consistent. For reference, an example of moth detection in a real scenario is shown in Figure 2.5 where codling moths are highlighted with red bounding boxes.

2.4.3 Network Optimization and Data augmentation

Much of the success of deep learning has come from building larger and larger deep neural networks. This allows these models to per-

form better on various tasks but also makes them more computationally demanding. Larger models take more storage space which makes them harder to distribute. Also, larger models take more time to run and can require more expensive hardware. The optimization phase aims to reduce the size of models while minimizing the loss in accuracy or performance, thus speeding up the inference step. The proposed implementation applies optimization both during the training phase and before the evaluation of the training model.

Data Augmentation. Training machine learning models, usually requires large and heterogeneous datasets to achieve good generalization capabilities. In our case, the amount of training data, which is represented by the number of training patches, is much smaller than standard small-scale image classification datasets [59, 60], which have on the order of 50,000 training examples. Therefore, we performed data augmentation to increase the number of images for training and incorporate invariance to basic geometric transformations into the classifier. Based on the "top-view" nature of the trap images, a certain patch will not change its class label when it is slightly translated, flipped, or rotated. Therefore, we apply these simple geometric transformations to the original patches to increase the number of training examples.

Pruning. Neural network pruning is a compression method that involves removing unnecessary neurons or weights from a trained model. There are different ways to prune a neural network. 1) Weights pruning by setting individual parameters to zero and making the network sparse; 2) removing entire nodes from the network, making the architecture smaller, while keeping the accuracy of the initial larger network. In our case, we prune weights so as not to impact the accuracy of the performance. Network pruning is executed iteratively during the training phase to achieve the desired accuracy during the validation phase.

Model Optimization. After the training phase, the model has been further optimized to reduce the network complexity and to increment the evaluation speed. To facilitate faster model inference, we perform Node Merging [61], Constant and Horizontal Fusion [62], Batch Normalization [63], and drop of unused layers (Dropout layer used during the training phase).

2.5 Results and Evaluation

This application checks the presence of codling moths twice per day. In one application cycle, the smart trap has to perform the following tasks:

- [**Boot**] - Power ON
- [**Task 1**] - Take a picture of the trapped insects
- [**Task 2**] - Pre-process the captured images
- [**Task 3**] - Execute the classification algorithm
- [**Task 4**] - Send the computation results using the radio
- [**Shutdown**] - Power OFF

These steps are used to characterize the smart trap performance in terms of power consumption and required energy. Moreover, since the system has preferably to work unattended indefinitely, a few remarks and considerations about its energy balance are taken into account.

2.5.1 Power consumption

Thanks to the external low-power microcontroller, the system can wake up only when planned. To characterize the system consumption, we measure the current required by each task by comparing twelve configurations:

- Raspberry Pi3 evaluating MobileNet V2;
- Raspberry Pi3 evaluating LeNet;
- Raspberry Pi3 evaluating VGG16;
- Raspberry Pi3 supported by NCS evaluating MobileNet;
- Raspberry Pi3 supported by NCS evaluating LeNet;
- Raspberry Pi3 supported by NCS evaluating VGG16;
- Raspberry Pi4 evaluating MobileNet V2;
- Raspberry Pi4 evaluating LeNet;

- Raspberry Pi4 evaluating VGG16;
- Raspberry Pi4 supported by NCS evaluating MobileNet;
- Raspberry Pi4 supported by NCS evaluating LeNet;
- Raspberry Pi4 supported by NCS evaluating VGG16;

Table 2.2: Battery recharge time for both a single application cycle and for fully charging the battery [20-100%] while harvesting solar energy at three different illuminance levels.

	2000 lx	**10000 lx**	**25000 lx**
Full Battery	60 h	13 h	7 h
Single Cycle	23 min	5 min	3 min

These configurations permit finding the software bottlenecks and selecting the best trade-off for the overall system performance. As can be noted in Table 2.3, Task 3 is the one that requires more energy because of the neural inference computation together with the use of NCS. Even if it boosts the neural inference step, its VPU requires a high power to perform accelerated edge inference. Here, the best arrangement that absorbs less power on average than others is the Raspberry Pi3 evaluating LeNet-5. Figure 2.10 shows the energy consumption of each task. In this case, Task 3 is confirmed as the most power-hungry activity and requires more energy than others. Moreover, configurations that involve Raspberry Pi4 are characterized by higher energy consumption than others. This is due to the Pi4, which features oversized CPU performance. Even if it executes the application faster than Pi3, it requires more energy to complete one application cycle. Thus, the configuration consisting of the Raspberry Pi3 running LeNet demonstrates to provide the best trade-off.

2.5.2 Expected Battery Lifetime

The platform is powered by a single-cell LiPo battery with 1820 mAh capacity. Considering the most energy-demanding configuration (i.e., Raspberry Pi4 evaluating MobileNetV2 consuming 200.1 J for one application cycle), the battery can supply the system for

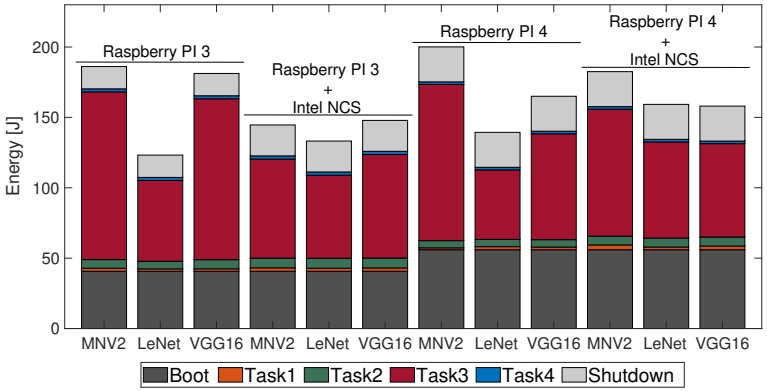
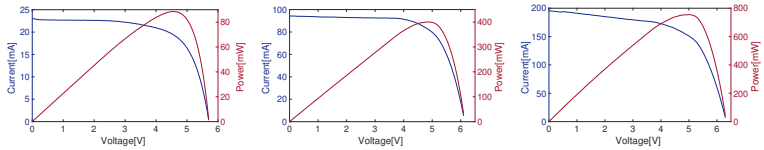


Figure 2.10: Tasks Energy consumption breakdown comparison for a single cycle of the implemented application. Single-task energy is presented in Table 2.3 for each implementation.

about 120 complete cycles, which lasts around two months. On the other hand, if the best configuration is considered (i.e., Raspberry Pi3 evaluating LeNet consuming 123.2 J for one application cycle), the system operates unattended for about 200 cycles (around 3 months) with the only battery.



(a) (b) (c)

Figure 2.11: Solar panel characterization respectively (a) at 2000 lx, (b) at 10 klx and (c) at 25 klx.

2.5.3 Energy Harvesting and Platform Sustainability

In apple orchards, energy resources are limited. Thus a self-sustainable system leads to a more robust and durable application to foster

farmers to "deploy and forget" the sensors. For this purpose, an energy harvesting system has been designed and characterized. It is composed of an MCU to trigger the power-on and shut down and a 140mm×100mm solar panel used for recharging the battery. The solar panel used has been characterized using several different light levels. Figure 2.11 present the I-V curves of the solar panel in three different environmental situations: 2000 lx (i.e., cloudy, Figure 2.11a), 10000 lx (i.e., fair, Figure 2.11b) and 25000 lx (i.e., sunny, Figure 2.11c).
By considering the measured power output from the solar harvester, we measured the time duration of the whole charge process of the battery in different conditions. We have measured both the time needed to charge a depleted battery and the time to harvest the energy necessary for a single application cycle. The results are presented in Table 2.2. Starting with the lower illuminance level, the recharging time is lower than the application duty cycle (i.e., 2 cycles per day), validating the sustainability of the platform. This feature is also confirmed by the graphs presented in Figure 2.12 that show the battery energy trend while harvesting with an illuminance equal to 7000 lx. It can be argued that even by emulating 3 days of cycles, the battery level does not drop, validating the hypothesis that the platform can self-sustain its operation thanks to the integrated solar harvester.

2.6 Conclusions

Computer vision systems are already widely employed in different segments of precision agriculture and industrial food production. Running deep learning features on the edge can optimize the management of fruit orchards.
This chapter presents a computer vision solution for automating pest detection inside orchards. The platform exploits ML functionalities on edge to evaluate images captured inside common pheromone traps to get early detection of dangerous parasites. Furthermore, on board inference avoids the transmission of the whole image, reducing the wireless communication bandwidth and energy costs. We analyzed the best hardware configuration using different neural networks, trained to get the best pest detection accuracy. Moreover, we combined a designed energy harvester to demonstrate the

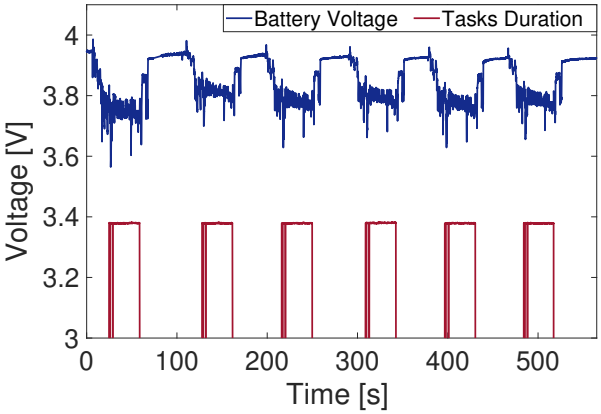


Figure 2.12: Platform sustainability evaluation. The graphs present the battery voltage trend while harvesting solar energy with a 7000 lx illuminance.

perpetual operation of the device unattended.

Table 2.3: Energy tasks breakdown of the tested platforms measured in J. The best trade-off is provided by Raspberry Pi3 evaluating the LeNet network.

Task Energy (J)	RPi3 MNV2	RPi3 LeNet	RPi3 VGG16	RPi3 MNV2 with NCS	RPi3 LeNet with NCS	RPi3 VGG16 with NCS
Boot	40.7	40.7	40.7	40.7	40.7	40.7
Task 1	2.171	1.674	1.733	2.427	2.177	2.359
Task 2	6.125	5.453	6.491	6.923	7.054	7.041
Task 3	119.1	57.4	114.3	70.34	59.04	73.53
Task 4	2.147	2.147	2.147	2.273	2.273	2.273
Shutdown	15.85	15.85	15.85	21.97	21.97	21.97
Total	**186.1**	**123.2**	**181.2**	**144.6**	**133.2**	**147.9**

Task Energy (J)	RPi4 MNV2	RPi4 LeNet	RPi4 VGG16	RPi4 MNV2 with NCS	RPi4 LeNet with NCS	RPi4 VGG16 with NCS
Boot	56	56	56	56	56	56
Task 1	1.327	2.179	1.907	3.385	1.934	2.609
Task 2	5.099	5.139	5.221	6.257	6.37	6.423
Task 3	111	49.39	75.22	90.2	68.28	66.33
Task 4	1.822	1.822	1.822	1.822	1.822	1.822
Shutdown	24.84	24.84	24.84	24.84	24.84	24.84
Total	**200.1**	**139.4**	**165**	**182.5**	**159.2**	**158**

Chapter 3

TinyML-based Autonomous UAVs

3.1 Automatic Landing in Resource Constrained UAVs

Unmanned Aerial Vehicles (UAVs) can operate autonomously in dynamic and complex operational environments and are becoming increasingly common. Deep learning techniques for motion control have recently taken a major qualitative step because vision-based inference tasks execute directly on edge. The goal is to fully integrate the machine learning element into the small UAV. However, given the limited payload capacity and energy available on small UAVs, the integration of computing resources sufficient to host ML, autonomy, and vehicle control functions is still challenging. This chapter [1]

[1] The work presented in this section has been published in the following papers:

- Albanese, A., Nardello, M., and Brunelli, D. (2022). Low-power deep learning edge computing platform for resource constrained lightweight compact UAVs. Sustainable Computing: Informatics and Systems, 34, 100725.
- Santoro, L., Albanese, A., Canova, M., Rossa, M., Fontanelli, D., and Brunelli, D. (2023, June). A Plug-and-Play TinyML-based Vision System for Drone Automatic Landing. In 2023 IEEE International Workshop on Metrology for Industry 4.0 and IoT (MetroInd4.0IoT) (pp. 293-298). IEEE.

presents a modular and generic system that can control the UAV by evaluating vision-based ML tasks directly inside the resource-constrained UAV. Two different vision-based navigation configurations are tested and demonstrated. The first configuration implements an autonomous landing site detection system. Two architectures are evaluated, namely LeNet-5 and MobileNetV2. This allows the UAV to change its planned path accordingly and approach the target to land. The second configuration presents a model for people detection based on a custom MobileNetV2 network. Finally, the application execution time and power consumption are measured and compared with a cloud computing approach. The results show the ability of the developed system to dynamically react to the environment to provide the necessary maneuver after detecting the target exploiting only the constrained computational resources of the UAV controller. Furthermore, it is demonstrated that moving to the edge, instead of using cloud computing, can lower the energy requirement of the system without reducing the quality of service.

3.1.1 Introduction

In recent years, considerable research has been conducted regarding the design, development, and operation of autonomous Unmanned Aerial Vehicles (UAVs). Multi-rotor UAVs are potentially useful in a wide variety of scenarios. They can work in extreme environments to monitor and explore areas hardly reachable by operators, and they can perform tasks such as rescue operations [64, 65], wild monitoring [66, 67, 68], and pest detection [69, 70, 71], telecommunications relay [72, 73, 74], border surveillance [75, 76, 77] and many others [78, 79, 80, 81, 82, 83].
While UAV control technologies have progressed steadily in recent years, the main control methodologies are still radio remote and preprogrammed. Nevertheless, these fields of application impose enormous constraints for normal operation tasks such as taking-off, navigation, object detection, environment interaction or landing. To overcome these limitations, researchers have proposed different approaches to ease each task [84, 85, 86, 87].

- Albanese, A., Taccioli, T., Apicella, T., Brunelli, D., and Ragusa, E. (2022, September). Design and deployment of an efficient landing pad detector. In International Conference on System-Integrated Intelligence (pp. 137-147). Cham: Springer International Publishing.

The growing amount of processing resources sufficiently portable for deployment on-board lightweight UAVs has made it possible to run machine learning-based image processing on these devices in real-time [88]. Machine learning models have been integrated into UAVs for many years, but only recently, Deep Neural Networks (DNNs) have been utilized for various autonomous UAV operations. DNN models exhibit "excellent capabilities for learning high-level representations from raw sensor data" [89]. Unfortunately, DNNs require training on large and specialized datasets to achieve good results in many applications [90] and getting these models to edge devices has been challenging.
Usually, UAVs have limited onboard computation resources, thus requiring a significant amount of time to process the acquired data and decreasing the UAV's responsiveness to the external environment. Moreover, continuously running complex algorithms imposes a considerable energy expense which can affect the operational lifetime. To overcome this limitation, researchers have proposed to exploit cloud computing to offload part of the computation [91, 92]. In the literature, we can find different architectures [93, 94] to relieve the resource constraints of UAV processing capabilities. These approaches enable UAVs to benefit from the redundant resources of modern data centres. However, even if cloud-based processing offloading has obtained some positive results [5], they have yet to achieve high Quality of Service (QoS) for resource-intensive applications mainly due to the following reasons:

- **Data transfer:** Data transmission from the UAV to the remote cloud can cause latency issues. This is because some UAV tasks are latency-intolerant. After all, they have to make near-real-time decisions, and even slight delays can cause dangerous consequences (i.e., collisions). Moreover, a large amount of streaming data causes long communication delays when the data is transferred to remote centralized data centres for processing.

- **Energy cost:** While computation offloading to cloud servers can reduce the energy consumption associated with data processing, it considerably increases the overall energy consumption due to the energy cost of transmitting the data. The problem can also worsen if the applications are communication-intensive or the UAV experiences poor network connectivity.

Therefore, in some conditions and locations, the time needed to transport data to the edge server and receive the analysis outcome may exceed the time with local processing at the UAV.
The challenge of optimizing resource utilization on resource-constrained devices has also been tackled in the setting of IoT computing. Researchers [95, 96, 97, 98] have proposed to move part of the data analysis near the sensor where the data is collected, by using the so-called edge computing approach. Edge computing permits to meet the processing requirements in a scalable, context-aware, and interoperable manner by enabling the distribution of data processing between devices and gateways intelligently, and between the cloud and endpoint devices. Nevertheless, UAVs have to be equipped with high-performance and ultra-low-power neural accelerators to perform learning model evaluation and meet the requirements of the latency-critical application.
This chapter presents a framework platform for enabling machine learning on the edge for the fast evaluation of learning models directly on-board UAVs. The proposed implementation exploits the computational power of Raspberry PI SBC equipped with a neural accelerator. By using a visual sensor, the UAV can promptly react to the changing environment in near-real-time. Two ML tasks are tested and characterized to evaluate the proposed solution: **1)** An autonomous landing algorithm based on the evaluation of a DNN; **2)** An autonomous navigation algorithm based on people detection and tracking. This chapter implements the following contributions:

- The assessment of edge computing is more profitable than cloud computing from an energy point of view by testing it with a real-time system (e.g., drone).
- The implementation and characterization of a processing platform for vision-based inference on the edge. Different HW configurations are tested to assess the best configuration from an energy processing point of view.
- The implementation and characterization of different learning models to assess the platform functionalities even with the execution of multiple DL models with different complexities in the same run-time.
- The implementation of an autonomous vision-based control mechanism for lightweight UAVs. The prototype developed

and used for testing the autonomous navigation functionalities is presented in Figure 3.1.

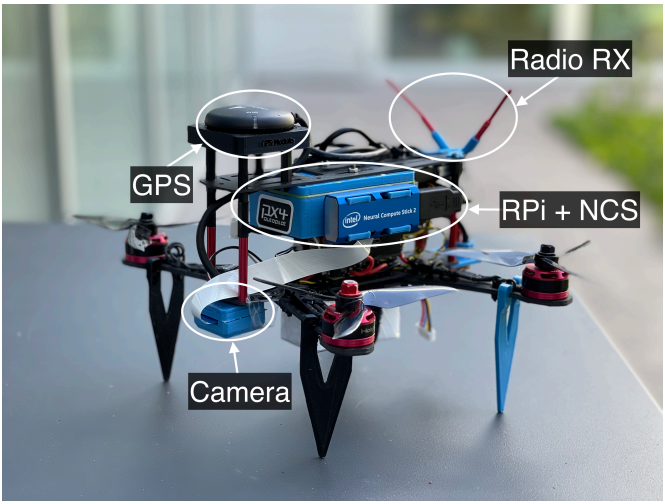


Figure 3.1: Lightweight UAV prototype.

3.1.2 Related Works

Advances in UAVs offer unprecedented opportunities to boost a wide array of IoT applications. Nevertheless, UAV platforms still face significant limitations, such as autonomy and weight, which impact their remote sensing capabilities. Therefore, capturing and processing the data required for developing autonomous and robust real-time object detection systems is still challenging.

To complete the scheduled mission, a UAV must be fully aware of its states, including location, navigation speed, heading direction, starting point, and target location. Various navigation methods have been proposed, and they can be mainly divided into three categories: inertial navigation [99, 100], satellite navigation [101], and vision-based navigation [102, 103].

Due to the rapid development of compact, low-power computer vision systems, vision-based navigation proves to be a primary and promising research direction for implementing autonomous navigation. However, UAVs constitute a specific case where the complexity

of vision tasks is substantially increased due to the rapid movements that such a platform can experience. These movements are not limited to lateral-forward but also involve movements along the vertical axis, which affect the visioned size (i.e., scale) of the tracked object [103, 104].
Recently, researchers have proposed to use ML algorithms to enhance autonomous UAVs' reliability and efficiency [105]. In [106] a system for safe landing area detection is developed. It combines classical computer vision and deep learning algorithms to improve system performance. Our work reveals that this combination is a successful approach to enhance the detection accuracy and optimize the execution time as well as the power consumption. The authors in [107] propose a two-stage training system to spot landing areas. The first stage consists of a CNN trained with synthetic images of landing areas. The second one employs a transfer learning approach during flight by using weights produced by the previously trained network. In this way, it is obtained a specialized model that can be used in different scenarios. In [108] a weed detection system for precision agriculture is executed autonomously directly on-board an Odroid-U3+ SBC. Results show how the SBC can compute the (x, y) target coordinates, colour detection, and weed detection using an object-based image analysis algorithm.
Machine learning methods can be divided into two categories based on the processing methods: Cloud (offline) and Edge (online) learning approaches.

Cloud Architecture

The missing hardware and platform able to run ML directly near sensors and the incoming of the Internet of Things (IoT) which permits the exchange of even big data through Internet nodes, has pushed to use the cloud inference approach. It can overcome the computational limitation problem that characterizes most robotic systems by moving the high-computing demand off-board. The idea consists of collecting useful data while flying and sending them via wireless LAN to a server that computes the inference and sends back the result. For instance, a cloud-based real-time object tracking by using UAVs called "Dronetrack" is developed by [109]. This system reveals its cloud architecture which exchanges data between drone, server, and user on the ground. In this way, it is possible to send

the video sensed by the drone and use it to perform object tracking off-board. This system is tested in three different scenarios: a walking person on a football pitch using a WiFi router operating with 3G/4G, a walking person in a city district with an optic fibre connection, and a moving vehicle in a city district with an optic fibre connection. Fast connections (e.g., optic fibre) can lead to a better performance thus reaching a real-time execution. However, the optic fibre connection cover area is limited, and it cannot be available in every type of environment and situation. On the other hand, the arrival of 5G can enhance the cloud architecture capability by improving its data transmission velocity. The object tracking system tests also revealed an additional limitation due to object velocity. Objects that are moving too fast, after all, are difficult to be tracked (e.g., a moving vehicle). Still, ML algorithms can overcome this issue and add more stability and reliability to the system.
A cloud-based autonomous system opens the usage of complex ML models. However, data transmission introduces a consistent latency that limits this approach to only a few application requirements. For example, considering an application where a drone has to interact actively and continuously with the surrounding environment, the cloud inference approach, usually, is not the best choice. Moreover, the authors in [110] have developed a cloud-based object detection system for UAVs using a Region-based Convolutional Neural Network (R-CNN). CNNs are characterized by an enormous number of operations (especially with the 2D convolution layers and with fully connected layers) to compute a prediction giving an input image. Thus, using a cloud inference approach makes it possible to move the high computing demand to servers with unconstrained resources. This system is tested and characterized by comparing the execution speed and the prediction accuracy with the state-of-the-art YOLO and SSD object detectors with aerial images. This test reveals that the proposed R-CNN presents the best accuracy (83.9 mAP) but the lowest speed (3.48 FPS). Furthermore, an additional comparison is conducted by considering the execution time of fast YOLO on a local laptop and the R-CNN on a remote server as a simulated cloud. In the first case, YOLO takes about 7 seconds to carry out one application cycle, while R-CNN takes only 1,29 seconds including the transmission latency which is one-third of the whole time.

Edge Architecture

Cloud systems have revealed their powerful aspects because of their almost unlimited resources. However, a wireless and robust connection is not always available, and the introduced transmission latency might be consistent, thus limiting UAV operations only in specific situations. Autonomous UAVs based on edge computing architectures avoid data transmission and can be scaled in every application by reaching better execution performance. Furthermore, edge computing is already widespread in fixed robotic systems (e.g., surveillance and monitoring) to slim down the amount of data that needs to be transmitted. For instance, a fixed robotics system for pest detection is developed in [10] [26] [111]. This research has implemented an ML-based system for pest detection in apple orchards using Raspberry Pi3 equipped with a Pi camera and Intel Movidius NCS. Here, the edge inference approach is used to process the data through a VGG16 CNN [112] for insect classification and obtain the result (i.e., the effective presence of dangerous insects). In this way, it is possible to decrease the amount of data to be transmitted to the user (e.g., apple farmer) by limiting it to a simple notification of a few bytes.
The proposed hardware can be scaled to mobile robotics applications to enable autonomous navigation. A perception, guidance, and navigation system for autonomous drone racing by using DL techniques is developed by [113] which navigates racing drones through gates. The most common computer vision techniques have been revealed inefficiently, but it is possible to estimate the gate centre quickly and accurately by using CNNs. This system uses NVIDIA Jetson TX2, - one of the most powerful boards to perform inference near sensors with impressive performance- as a computer on-board to carry out all vision processing. The presented system is compared with other two state-of-the-art networks (VGG-16 based SSD [114] and AlexNet based SSD [115] [116]), resulting as the fastest (28.95 FPS) but the worst in terms of detection accuracy (85.2% detection rate). The NVIDIA Jetson TX2 is also used for embedded real-time object detection in a UAV warning system [117]. It uses the state-of-the-art Yolo V2 [118] network which enables dangerous product recognition directly on-board. Specifically, the Jetson TX2 is responsible for the execution of the object detection algorithm and positioning algorithm. At the same time, the decision sup-

port engine back-end is responsible for the warning alarm if people are getting too close to dangerous areas. In this case, the system performance is evaluated in terms of image input resolution and detection frame rate by comparing Yolo V2 [118], and Tiny Yolo [119]. This research confirmed that using low-resolution input images and simple network architectures (e.g., Tiny Yolo) can lead to faster execution.
Edge processing needs the usage of model compression techniques to better exploit the power of DNNs in edge platforms. The challenge consists of highly compressing models without losing accuracy; in this way, it is possible to move intelligence in mobile devices or IoT systems by keeping the service quality [120]. The most used compression techniques are quantization and pruning; the first quantizes weights from floating point to integer to reduce the number of bits and, consequently, the model memory footprint. The second one cuts off weights and their relative connections that are close to 0 to reduce the model complexity. This operation usually needs to retrain the model to fine-tune it with the missed branches. On the other hand, these techniques can lead to optimized model complexity, but, they can reduce the service quality by losing prediction accuracy. Surveys have shown that even if the model is highly compressed, the loss in accuracy is negligible (i.e., in the order of 0.001 %) [121] [122] [123] [124] [125]. By using model compression techniques, it is possible to have a lightweight system optimized to perform real-time inference in edge devices such as embedded GPUs, SBCs (e.g. Raspberry) or MCUs. Nevertheless, these devices have limited memory, thus the model compression step is fundamental. For instance, embedded GPUs and SBCs can host models of GBs, but MCUs have a few hundred KBs available. This sets the platform limits and its potential to run complex ML algorithms [126].

Platform selection

Many researchers are looking to find the best trade-offs in using DL techniques for resource-constrained environments such as UAVs. DroNet [127] is an efficient CNN detector for real-time UAV applications. It uses network optimization and pruning to optimize the memory footprint of a modified Tiny Yolo architecture. The improvements consist of using a single class classification and changing the number of filters, layers, input size, convolutions, and pool-

ing layers. With the proposed modifications, the new architecture reduces the feature map by a factor of 2. The system performance is assessed with a vehicle detection application on an Odroid-XU4 board which reveals a frame rate execution of around 8-10 FPS with an accuracy of 95%. Additionally, the same system is moved to Raspberry Pi3, which reveals worse performance due to the less capable CPU. As shown, DroNet is suitable for drones' real-time applications, however, it is based on a cloud architecture: the drone streams a video to a base station that performs the processing and sends back the navigation commands. EdgeNet [128], instead, efficiently performs object detection with the edge computing approach. It is composed of three steps; the first consists of a lightweight CNN that detects objects through a simple CNN for object detection. It follows the selection of image regions that contain the detected objects to reduce the image size and, thus, the further processing complexity. Finally, it employs an optical flow-based tracker to further reduce the processing demand and track the detected objects without using the CNN-based object detector. EdgeNet can process high-resolution images, such as aerial images, with a low-power profile and a fast processing execution, thus making it suitable for UAV applications. However, the presented work use-case shows the feasibility and the cost-efficiency of using edge architecture for drone applications, thus performing all processing on-board on drones and reducing the processing demand.
FPGA-based platforms can lead to better performance in terms of energy consumption and inference acceleration, however, its usage is strictly tailored for a specific application, thus not scalable for related applications that, for instance, may require additional sensors [129]. SBCs such as Raspberry, Odroid and BeagleBone are more flexible than FPGA-based platforms and they can enable edge computing by processing raw sensor data. Embedded GPUs such as NVIDIA Jetson Nano and Google Coral are widely used in edge computing as well. On the other hand, they lead to fast execution of neural inference but with high power consumption (in the order of 10 W). In this use case, embedded GPUs are oversized because the used neural models are not so complex to need a GPU for real-time execution. This reason has moved the attention to the SBC family. Odroid and BeagleBone feature high execution performance, but they are still too energy-demanding. Moreover, they are not supported by a large community as Raspberry Pi is, thus making

them less affordable for prototyping. In addition, the camera used in this case study is not supported by Odroid and BeagleBone. Raspberry presents 40 i/o pins and the CSI camera port. It perfectly fits these application requirements for connectivity, supported modules, execution performance, and power consumption [130].

3.1.3 Video Processing

Video pre-processing is a fundamental step in AI vision-based applications to make correct predictions and train a CNN. A basic video processing algorithm highlights relevant features of frames (e.g., contours, edges, contrast) and reduces the noise introduced by the image sensor. This section introduces the video processing algorithm developed to create a consistent dataset for CNN training in a semi-automatic way. The same algorithm is used as a pre-processing step in the final application workflow. For this purpose, many videos of different landing pads in different environmental situations are acquired to compose a heterogeneous dataset (e.g., landing pad in a garden, on the street, in a car parking).

Pre-processing implementation

The algorithm is implemented by following a particular approach for video pre-processing for AI applications. It consists of noise reduction, features highlighting, and cropping of the region of interest (RoI) which may contain a landing pad.

Figure 3.2b presents the algorithm workflow and it works as follows:

- **Read video**: the video can be acquired offline and then read to retrieve each frame; otherwise, it can be streamed from a camera (e.g. the Pi camera) and read from Raspberry Pi.
- **Denoise**: it consists of a Gaussian blurring of the image to filter out high-frequency noise. A kernel mask of shape 9×9 is used.
- **Gray scale**: conversion from RGB to gray-scale.
- **Segmentation**: image binarization through an adaptive Gaussian threshold.

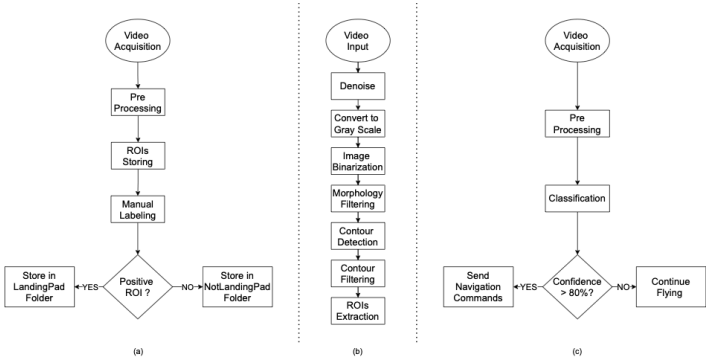


Figure 3.2: Machine learning task workflow. (a) Dataset generation flowchart. (b) Pre-processing algorithm flowchart. (c) Landing pad detection algorithm flowchart. Flowchart (b) presents the preprocessing block of flow chart (a) and (c)

- **Morphology**: application of morphological filters with a structuring element of size 3×3 and unitary element values. Two iterations of opening and four iterations of dilation are applied to highlight each blob of interest.
- **Contour detection**: this step finds each contour revealed from each processed frame.
- **Contour filtering**: filter out small contours (filter by area) and contours that do not present any hierarchy level (filter by hierarchy).
- **Crop RoI**: each region of interest around each filtered contour is extracted.

Then, each cropped RoI can be used as:

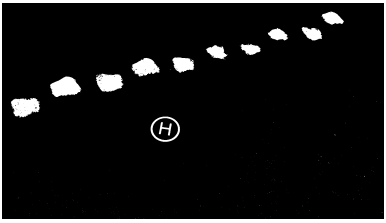
- **NN input**: before a resizing for matching the NN input shape, RoI is used in the final application as NN input.

- **Dataset**: RoI is used to construct a large dataset for the NN training.

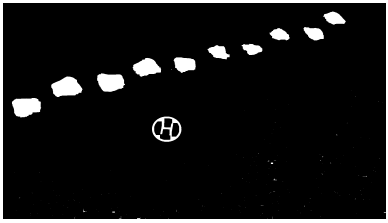
The step-by-step result of the video processing algorithm is summarized in frames in Figure 3.3. The algorithm implementation is

(a) Original frame.

(b) Gaussian blur.



(c) Binarization.



(d) Morphological filtering.

(e) Contour extraction and filtering.

(f) Cropped RoI.

Figure 3.3: Example of video processing algorithm workflow.

carried out by using *Python* program language and the *OpenCV* library.

3.1.4 Landing pad Detection

The landing pad detection problem can be expressed in terms of symbol/character recognition and image classification. Therefore, a modified structure of the LeNet-5 [131] [18] CNN architecture, presented in Figure 2.6, is used. It has been initially developed for character recognition, but it can be extended to classification problems, especially for this application purpose that involves the detection of symbols on a landing area. It presents a straightforward architecture, which makes it feasible for embedded applications.

This NN takes gray-scale images (i.e. single channel) of size 64×64 as input which is processed through two 2D convolution layers with a *ReLU* activation function (despite to the original structure where it is used a *tanh* activation function) which exhibits a better classification accuracy than *tanh*. Furthermore, the second convolutional block does not use all the features extracted by the average pooling layer, thus permitting learning different patterns and enhancing the classification accuracy. This peculiarity makes the network even less computationally demanding, and thus suitable for embedded platforms. Finally, the last layer is a *Softmax* classifier which provides the confidence probability among two classes (binary classification): *Landing pad* and *Not landing pad*. Moreover, the state-of-the-art MobileNetV2 [2] [132], presented in Figure 2.8, is trained with the same dataset to compare its performance with LeNet. It presents a more deep and complex architecture than LeNet, but, due to the inverted residual block, it drastically reduces by a factor of 13 the number of parameters compared to LeNet. On the other hand, LeNet uses a fully connected layer to flatten the output which considerably increases the network number of parameters.

Dataset construction and organization

The dataset construction and organization are important steps for NN training. It is preferable to have a consistent and heterogeneous dataset to obtain an accurate model with generalization capabilities. For this purpose, the dataset is constructed by using the algorithm explained in section 3.1.3 which processes multiple videos of different kinds of landing pads acquired offline. In this way, it is possible to generate a dataset with thousands of images in a semi-automatic way by following the workflow in Figure 3.2a. However, not all ob-

tained RoIs represent landing pads, and sometimes the algorithm shows RoIs with miscellaneous objects such as storm drains, tiles, and plants. Considering that the pre-processing algorithm may fail in this way, it is chosen to use this kind of object to construct the against class *Not landing pad*. By doing so, CNN can use images that could be truly the network's input.
Furthermore, synthetic images -generated by [133]- representing landing pads with different shapes and symbols are added to the dataset to obtain a more heterogeneous one. The dataset is organized in folders to facilitate the automatic label extraction of each image. It consists of 13576 images for the *Landing pad* class and 12807 images for the *Not landing pad* class. The obtained dataset is composed of various types of landing pads and many miscellaneous objects. Figure 3.4 shows a few examples of them.

CNN training

The final application requires the trained CNN deployed on the Raspberry Pi micro-computer to perform edge computing and reach high performance. Thus, it is necessary to focus on the memory footprint occupied by the neural model. For this purpose, many training sessions with different parameters (i.e., number of epochs, input image size and depth, and batch size) are executed. The best trade-off is found with parameters shown in table 3.1 and performing data augmentation to enhance the dataset and the network accuracy.

Table 3.1: Landing pad detection algorithm training parameters.

	Network Architecture	
	LeNet-5	**MobileNet V2**
Epochs	20	20
Batch size	32	16
Initial learning rate	10^{-3}	10^{-3}
Input image	64×64×1	100×100×1
Optimizer	Adam	Adam
Loss function	Cross-entropy	Cross-entropy
Source framework	Tensorflow	Tensorflow
Training accuracy (%)	99.8	99.9
Validation accuracy (%)	99.2	99.9

(a) Hand crafted. (b) Blue landing pad. (c) Red landing pad.

(d) CiterX pad (synthetic). (e) Green square pad (synthetic). (f) Red landing pad (synthetic).

(g) Storm drain. (h) Tile. (i) Bike.

Figure 3.4: Example of dataset images.

CNN test

Both networks are tested with about 3000 images that are retrieved from the dataset before the training session. Moreover, a few images of even different landing pads found on the web (about 30) are added to the test dataset to assess the network generalization capability. The results are summarized in Table 3.2. Both networks reach satisfying results in accuracy and precision, thus making these neural models suitable for this application purpose. An example of

Table 3.2: LeNet and MobileNetV2 test results.

(%)	Accuracy	Recall	Precision	F-score
LeNet-5	99.4	99.5	99.9	99.7
MobileNetV2	99.4	99.6	100	99.8

landing pad detection on a test aerial video is shown in Figure 3.6a.

Embedded implementation

The overall algorithm has to run on the Raspberry Pi single-board computer equipped with a Pi Camera and the Movidius NCS, which accelerates and optimizes the neural inference step. Furthermore, the *OpenVINO* toolkit is used to optimize the landing pad detection model to an Intermediate Representation (IR) and perform video processing and neural inference on the Movidius NCS.
The landing pad detection algorithm workflow is shown in Figure 3.2c, and it consists of the following steps. The Pi Camera enables a continuous streaming mode at 30 FPS, and each frame is retrieved. Then, the frame is processed with the pre-processing algorithm shown in section 3.1.3 which outputs RoIs that may contain a landing pad. Because the pre-processing algorithm is not reliable as a landing pad detector, each RoI is classified with the CNN presented in section 3.1.4, which gives an accurate result by identifying the effective presence of a landing pad. The classification step produces a confidence level that reveals how confident the CNN is that the input RoI is a landing pad. For this purpose, a threshold value of 0.8 is set to have the best precision and recall trade-off. It is possible to notice that the pre-processing step involves the usage of Raspberry together with Movidius NCS. In contrast, the classification step involves only the Movidius NCS, which boosts neural inference by reducing its execution time.

Execution of multiple DL models

This chapter addresses the problem of the fast deployment of DL models in resource-constrained environments such as UAVs. Thus, the developed framework platform capability is assessed by running multiple DL models in the same runtime. For this purpose, we use

and test a pre-trained model for people detection[1] (specifications in the Table 3.3) based on MobileNetV2-like backbone [2] [132].

Table 3.3: People detection model specifications.

Parameter	Value
Average Precision (AP)	88.62%
Pose coverage	Standing upright, parallel to image plane
Support of occluded pedestrians	YES
Occlusion coverage	<50%
Min pedestrian height	100 pixels (on 1080p)
GFlops	2.300
MParams	0.723
Source framework	Caffe
Input shape	$[1 \times 3 \times 320 \times 544]$
Color order	BGR

This model is executed in the target hardware together with the landing pad detection model. As shown in Figure 3.5, they work as two subordinate systems. When the drone is flying, it first checks the presence of people and, if recognized, it plans a path to avoid collisions with them. If any person is detected, it looks for a safe landing area (i.e., execution of the landing pad detection algorithm) and, if the confidence is greater than 80%, it autonomously lands on the detected landing pad. The pre-trained model used for people detection is suited to recognize pedestrians from an aerial survey, as shown in Figure 3.6b. However, due to the dataset used for the training session, the model needs a minimum pixel-wise pedestrian height of about 100 pixels on a 1080p image. This makes high-altitude people detection difficult. It is thus clear that, if the drone flies too high, it cannot achieve the desired detection accuracy. To enhance the people detection algorithm capability, it is possible to further train the model with a sparser dataset with aerial images representing people of different sizes and positions (e.g., people close or very remote to the camera system). Moreover, a transfer learning approach can be considered to improve the overall accuracy. It starts from an already trained network for related tasks and fine-tunes it

[1]Source: `https://docs.openvinotoolkit.org/latest/omz_models_model_person_detection_retail_0013.html`

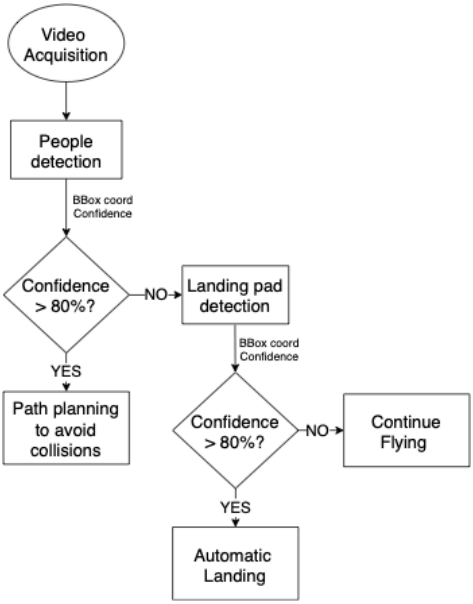
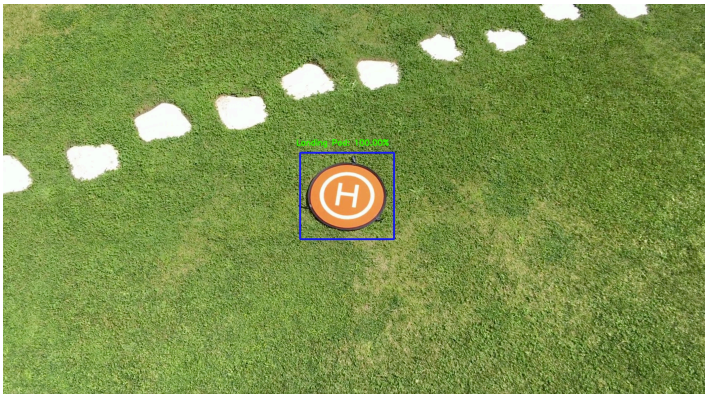


Figure 3.5: Execution of multiple DL models flowchart.

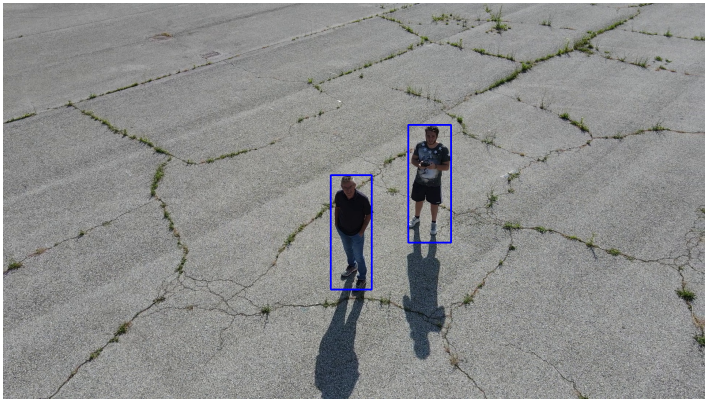
to obtain one with the required detection accuracy. For instance, it is possible to use a base network for people detection and extend its capability to detect very small people (i.e., a dozen pixels). By doing so, this model will not be limited by the people's height, and it will be suitable for aerial applications.

3.1.5 Results

The presented framework platform is characterized by measuring the execution time and power consumption by running the landing pad detection algorithm together with the people detection model. In this way, it is possible to find out the system bottlenecks and improve the platform. Furthermore, two hardware configurations are used to compare their performance: Raspberry Pi3 with Movidius NCS v2 and Raspberry Pi4 with Movidius NCS v2. For this purpose, the application workflow, referred to Figure 3.2c, is split



(a) Landing pad detection example.



(b) People detection example.

Figure 3.6: Demo of the proposed algorithms for people and landing pad detection.

into two general tasks:

- **Task 1**: Video pre-processing.
- **Task 2**: Neural inference.

The tasks referred to the video streaming and output post-processing (i.e., sending of the land command) are neglected because they are executed within a period hundreds of times less than the other two tasks, thus not affecting the measurement results. The measurement campaign is conducted by processing 50 frames acquired from the camera installed in a static scene with a landing pad and a few people. Then, the measurements are averaged, and each task's energy consumption and execution time are retrieved. Different configurations are considered to get a complete comparison of the two target hardware:

- Landing pad detection with LeNet.
- Landing pad detection with MobileNetV2.
- People detection and landing pad detection with LeNet.
- People detection and landing pad detection with MobileNetV2.

Landing pad detection on Raspberry and NCS

The execution of a single DL model is initially considered. Figure 3.7 shows the energy consumption among the different hardware and neural network configurations for one application cycle (i.e., the processing of one frame). Furthermore, Table 3.4 summarizes the required energy for each task and the relative execution performance.
The most power-hungry task is the video pre-processing which uses ad-hoc video processing techniques with high computational demand. Furthermore, the MobileNetV2 architecture increases the power consumption in the second task because of its more complex architecture than LeNet. By considering the energy consumption and the execution performance, the best configuration is Raspberry Pi4 evaluating LeNet architecture for landing pad detection. It consumes 0.246 J for processing one frame, and it reaches real-time performance with a processing execution of 22.26 FPS.

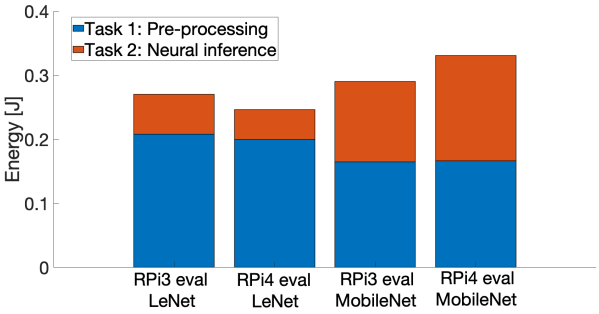


Figure 3.7: Landing pad detection energy comparison.

Table 3.4: Energy tasks breakdown and execution timing of the tested platforms by executing the landing pad detection algorithm alone. The best trade-off is provided by Raspberry Pi4 evaluating LeNet boosted with NCS.

Configuration	Task 1 (J)	Task 2 (J)	Total (J)	Elapsed Time (ms)	FPS
RPi3 eval LeNet	0.208	0.062	0.27	74.69	13.37
RPi4 eval LeNet	**0.2**	**0.046**	**0.246**	**44.90**	**22.26**
RPi3 eval MobileNetV2	0.165	0.125	0.29	76.73	13.04
RPi4 eval MobileNetV2	0.166	0.165	0.331	59.59	16.78

People detection and landing pad detection on Raspberry and NCS

The same comparison of the section above is conducted by running multiple DL models in the same run-time. In this case, the landing pad detector is running together with a people detection model to

set the platform limits. Even though both models share the same sensed data and the same NCS where the neural inference is performed, they are executed in a subordinate way and work as two independent systems. In Figure 3.8, it is possible to evaluate the energy consumption among the different configurations. Table 3.5 summarizes the required energy and execution performance.

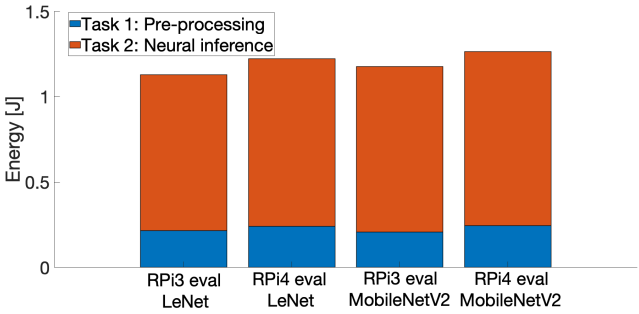


Figure 3.8: People and landing pad detection energy comparison.

Here, the most power-hungry task is the second one because of the people detection algorithm, which requires an extremely high computational power due to its large input image size and complex architecture. Moreover, the first task energy consumption is slightly increased compared to the previous case because of an additional standard pre-processing algorithm required by the people detector. If it is considered the energy consumption, the best configuration is Raspberry Pi3 evaluating LeNet which requires about 100 mJ fewer than the other configurations. However, its execution performance is degraded (3.23 FPS) due to the less capable chipset of Raspberry Pi3. On the other hand, when the execution time matters more than the energy consumption, the best configuration is Raspberry Pi4 evaluating LeNet. It uses 1.223 J to process one frame, but its execution performance is increased to 4.68 FPS.

Table 3.5: Energy tasks breakdown and execution timing of the tested platforms by executing the landing pad detector together with the people detection model. The best trade-off by considering the energy consumption is provided by Raspberry Pi3 evaluating LeNet boosted with NCS. However, if the execution time is considered, the best trade-off is provided by Raspberry Pi4 evaluating LeNet boosted with NCS.

Configuration	Task 1 (J)	Task 2 (J)	Total (J)	Elapsed Time (ms)	FPS
RPi3 eval LeNet	0.216	0.912	**1.128**	309.8	3.23
RPi4 eval LeNet	0.240	0.983	1.223	213.88	**4.68**
RPi3 eval MobileNetV2	0.207	0.969	1.176	322.86	3.1
RPi4 eval MobileNetV2	0.244	1.02	1.264	226.33	4.42

Drone flight time reduction

The presented platform aims to make UAVs autonomous; thus, the combination of Raspberry, Movidius NCS and the camera has to be mounted on-board on drones and plugged into their energy source (i.e., battery). However, the available energy in drone systems is limited, and it is a precious resource to ensure a long enough flight time. Therefore, the drone's battery has to power motors, ESC and electronic boards for flight control and, in addition, the RPi SBC which runs the autonomous navigation algorithms. For this reason, it is evaluated the degradation of the flight duration while using the proposed system.

We consider a 250 mm class drone (i.e., small size) equipped with a 1500 mAh battery operating at 14.8 V which ensures 15 minutes of flight in normal conditions. Furthermore, we consider the FPS's best configuration (i.e., Raspberry Pi4 evaluating LeNet) of the proposed system for estimating the drone battery lifetime. Table 3.6 compares the flight time and the corresponding energy for three arrangements: normal mode, execution of single model and execution of multiple models. As a result, the proposed platform reduces the drone flight time by only 6% for the execution of a single model

Table 3.6: Drone flight time reduction overview. Columns show the available flight time and the corresponding energy. Rows consider three different arrangements: without the autonomous navigation system (**normal**), with the landing pad detector (**single model)** and with people and landing pad detector (**multiple models).**

	Flight Time (min)	**Available Energy (KJ)**
Normal	15	80
Single Model	14.1	76
Multiple Models	14	75.6

and multiple models, thus it does not affect the battery degradation consistently.

Edge computing energy sustainability

Recently, researchers have highlighted that the edge computing approach outperforms the cloud one by avoiding data transmission and speeding up the application execution. To show how the energy consumption is distributed among the two different approaches, we compared the best configuration investigated above for landing pad detection (i.e. RPi4 evaluating LeNet) with the same application running on a Parrot Mambo mini-drone in cloud mode. Because the landing pad detection model and the test images are the same, the detection accuracy does not change; it changes the platforms where the application is executed, thus their performance in terms of energy consumption and execution time. The mini-drone utilizes a laptop as a ground station which features an intel core i5-3360M CPU@2.80GHz x 4. It is connected to the mini-drone with a private Wi-Fi network which enables data exchange among the mini-drone and ground station. The mini-drone acquires an image from its ground camera and sends it to the station; then, it performs image pre-processing (task 1) and neural inference (task 2) and sends back the landing command if a safe landing area is detected. Considering cloud computing, it is worthwhile to analyze the energy consumed from the drone side for networking overhead without considering the processing on the ground station because it is plugged into an un-

limited energy source. In this way, it is possible to find out the most cost-effective approach between cloud computing and edge computing and assess the platform's energy sustainability. The comparison is shown in Figure 3.9, and it demonstrates that the networking overhead for data exchange in cloud mode is much more expensive (i.e., 2.16 J) than the overall processing with edge computing (i.e., 0.246 J). This confirms the energy sustainability of the edge computing approach compared to the cloud one. Moreover, edge computing outperforms cloud computing considering the execution time. The first one takes 45 ms to process one frame, while the second one needs 1.4 s. This comparison clearly shows that edge computing satisfies real-time requirements in a resource-constrained environment with low energy consumption. Cloud computing, rather, degrades its performance due to the transmission latency (i.e., 1.34 s) and the required energy for data exchange.

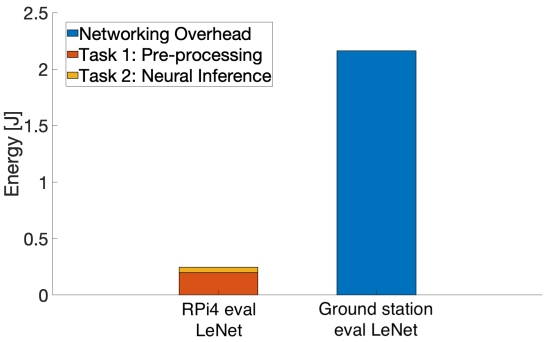


Figure 3.9: Energy consumption comparison of edge computing and cloud computing for landing pad detection.

3.1.6 Conclusions

Deep learning algorithms together with computer vision techniques are widely spread in autonomous UAV vision systems. However, performing autonomous applications among different constraints in performance, such as real-time execution, is still challenging. This chapter addresses this problem by implementing a platform for the fast deployment of DL algorithms directly onboard UAVs by using an edge computing approach. In this way, all computations and neural inferences are performed near-sensor, thus avoiding the big data management that characterizes cloud architectures. The platform performance is assessed with an autonomous landing algorithm that exploits ML functionalities. It is trained and compared among two different CNN architectures: LeNet and MobileNetV2. An additional pre-trained model for people detection is used and deployed on the proposed platform to characterize and compare its performance by running even multiple models in the same run-time. The best hardware and NN architecture configuration is found by considering the fastest and most low-energy execution. Final remarks about the energy impact of the platform on the drone flight time are taken into account. In future work, networks such as DroNet [127] and EdgeNet [128] are trained over the same dataset to compare them with the proposed solution and find the best cost-efficiency way to enable autonomous navigation.

3.2 Autonomous UAV Visual Odometry in Adverse Rainy Conditions

Reliable navigation is critical for autonomous drones performing life-saving missions in harsh weather conditions. However, rain significantly degrades their visual odometry-based navigation, potentially compromising mission success. This work [2] proposes a system to estimate rain severity, ensuring reliable navigation and data acquisition even during downpours. We analyze the performance degradation of visual odometry in various rain conditions, with the worst-case scenario yielding a 1.5 m average error in trajectory estimation. To address this challenge, we compare three deep neural network architectures for rain severity classification and optimal counteraction selection, for example, slow down, switch navigation systems, and land. The most lightweight network, reaching an accuracy of 90% with a memory footprint of 1.28 MB and a frame rate of 93 FPS, is ideal for real-time processing on resource-constrained drones.

3.2.1 Introduction

Drones equipped with depth cameras for visual navigation have become indispensable for critical missions like search and rescue, infrastructure inspection, and disaster response. These time-sensitive operations require immediate action and collaboration among multiple drones, making reliable navigation fundamental in all weather conditions. Unfortunately, visual odometry (VO) systems, crucial for drone navigation, are significantly affected by rain. Rain disrupts the visual scene, leading to inaccurate positioning and motion estimation, thus potentially jeopardizing mission success. It degrades the visual scene by altering image contrast, causing significant blurring due to water droplets on the camera lens, and masking crucial visual landmarks. In worst-case scenarios, such errors can cause mission failure, collisions, or even safety hazards. Therefore, developing methods to analyze and estimate rain severity's impact on VO accuracy is critical to ensure reliable UAV navigation in adverse weather.

[2]The work presented in this chapter has been conducted during a visiting research period at the Dyson School of Design Engineering of Imperial College London, United Kingdom, from September 2023 to February 2024.

Existing research on rain impact focuses on autonomous vehicles, where camera images are captured from the cockpit with water droplets on the windshield, which differs significantly from the direct lens exposure experienced by drones [134, 135, 136]. To address this gap, we investigate rain's impact on a VO system designed for drones in a laboratory setting that simulates a low-altitude flight. Building upon the success of deep learning (DL) for rain severity estimation in vehicles, we leverage these algorithms to improve the system's performance and reliability [137, 138]. However, unlike vehicles with substantial computational resources, small drones have limited capabilities. Therefore, designing the DL system to be aware of the on-board resource constraints is fundamental.
To overcome the challenges of rain on drone navigation, this chapter proposes a novel system that analyzes the performance degradation of a VO system under various rain intensities in a laboratory setting. A laboratory setup with a depth camera, processing unit, and rain simulator mimics a low-altitude drone flight. During these experiments, we collected a dataset with images of different rain intensities. Based on this analysis, we propose a DNN-based system for classifying and estimating rain severity from camera images captured by the UAV. The DNNs have been developed with a low-complexity impact, enabling deployment on resource-constrained edge devices like small drones [139]. By accurately estimating rain severity, the system enables the implementation of appropriate counteractions to maintain reliable navigation performance during UAV operations in rainy conditions, ultimately improving mission success rates.
Our chapter contributions include:

- A comprehensive dataset of approximately 335,000 real images categorized into 7 classes representing different rain intensity levels, ranging from clear to slanting heavy rain. This dataset is acquired through laboratory experiments simulating low-altitude drone flight.
- VO system characterization under varying rain conditions, quantifying average error, restoration time, and informing the selection of optimal counteractions in adverse weather conditions.
- The training, testing, and comparison of three state-of-the-art deep neural networks to identify the best trade-off between

performance and resource requirements for the deployment on resource-constrained drones.

3.2.2 Related Works

The rapid proliferation of commercial drones, from high-end aerial photography tools to delivery services, has made UAVs a ubiquitous sight in the skies. Despite their growing presence, a significant limitation remains: adverse weather conditions. While these versatile devices excel in various applications, rain, strong winds, and other weather events can affect them, hindering their effectiveness in real-time applications such as search and rescue. Unfortunately, current UAV systems are not equipped to handle all weather conditions, limiting their usefulness in critical scenarios [140, 141, 142, 4, 143, 144]. The research presented in [145] investigates "drone flyability", defined as the percentage of time drones can safely operate. The study reveals that common drones have a limited flyability of less than 5.7 hours per day (or 2.0 hours during daylight). However, this estimate excludes the impact of various adverse weather conditions, including rain, snow, fog, and high winds. The analysis highlights that enhancing weather resistance can dramatically increase flyability, with weather-resistant drones achieving up to 20.4 hours per day (or 12.3 hours during daylight). These findings underscore the fundamental role of weather in drone operations.
A crucial aspect not considered is the impact of adverse weather on a drone's autonomous navigation system. Rain, snow, fog, and high winds can disrupt the performance of sensors and navigation systems, potentially leading to malfunction and mission failure. This highlights the need for researches that address not only weather resistance but also the robustness of autonomous navigation systems in challenging weather conditions.
Deep-reinforcement learning holds promise for improving the reliability of autonomous UAV navigation within the Internet of Things (IoT) framework [146, 147]. However, a critical gap exists in researches that analyze the impact of adverse weather on these systems. Existing contributions often examine weather effects from an autonomous driving perspective, although, the sensing systems and datasets used in these studies are specifically designed for vehicles, making them unsuitable for small drones [148, 149]. While these studies focus on vehicles, they demonstrate the significant influ-

ence of weather on sensor performance. This suggests that similar effects likely impact UAVs, warranting investigation using comparable methodologies to develop robust autonomous navigation systems for UAVs operating in adverse weather conditions [150, 151, 152]. For instance, wind can disrupt flight stability, rain can obscure vision sensors, snow can accumulate on wings, fog can limit visibility, and flares can interfere with onboard electronics. For example, some researchers have investigated the impact of external wind on autonomous drone navigation. They have proposed a reinforcement learning-based solution to address these unknown disturbances, paving the way for the safe operation of autonomous UAVs in windy conditions [153, 154, 155]. However, similar solutions are needed to address the broader spectrum of adverse weather conditions that can compromise reliable navigation.

Rain is one of the most prevalent adverse weather conditions affecting UAV operations and its impact on visual odometry (VO) systems can be severe. Raindrops cause various perturbations to the sensing and perception system, including blurring due to scattered light, distortion from refraction, and reduced contrast due to reflection on the camera lens. These effects lead to pixel value fluctuations in captured images, ultimately causing inaccurate processing of raw data by VO algorithms. Deep learning-based algorithms, commonly used for object detection and image classification in UAVs, are particularly susceptible to rain, rendering them unreliable. Therefore, developing methods to assess rain severity is crucial for optimizing image processing and computer vision algorithms used in VO systems. This will ensure reliable navigation performance for UAVs even in rainy conditions [156].

While de-raining methods offer a post-processing approach to mitigate rain effects, their effectiveness can be limited. These techniques may not always deliver optimal performance and can significantly increase the computational burden on resource-constrained UAVs [157, 158, 159]. Other researchers propose specialized algorithms that outperform standard methods in adverse weather. However, these solutions are limited in scalability as they are often tailored to specific applications. These limitations highlight the need for a more comprehensive solution that addresses rain's impact on VO systems while maintaining computational efficiency and broad applicability across various scenarios [152, 160].

The adverse weather impact, in particular rain, represents a re-

search gap which makes this work a crucial study to implement autonomous UAVs that can operate 24/7.

3.2.3 Experimental Setup

To quantify the performance degradation of the visual odometry system under varying rain intensities, we designed a controlled laboratory environment. A key component of this setup is a custom-built water-resistant enclosure (Figure 3.10) housing the VO system. The VO system itself comprises a processing unit (Intel NUC 11 Pro Kit NUC11TNKi7) and a depth camera (Intel RealSense D435i). While these components are not inherently water-resistant, the enclosure ensures their protection during the experiments. The enclosure, constructed from polymethyl methacrylate (acrylic) panels cut using a laser cutting machine, measures $20 \times 15 \times 10$ cm ($length \times width \times height$), providing ample space for the VO system and necessary cables. For easy access to the internal components, the enclosure features a sealed lid at the back. An IP68-rated nylon gland further ensures water resistance by providing a secure connection point between the device's power supply and an external power source. The enclosure itself is assembled using a combination of bi-component glue, silicone sealant, and rubber seals, guaranteeing a watertight environment for the VO system.

The processing unit within the water-resistant enclosure executes

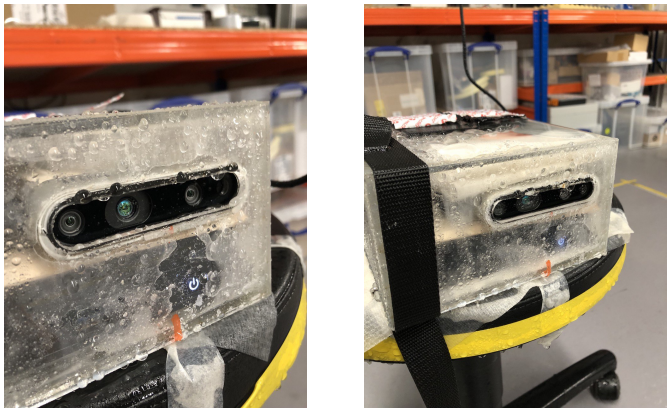


Figure 3.10: The water-resistant enclosure hosting the VO system.

the VINS-Fusion algorithm [161, 162, 163, 164]. It is an optimization-based multi-sensor state estimator suited for accurate SLAM (Simultaneous Localization and Mapping) in autonomous navigation applications. VINS-Fusion offers flexibility by supporting various visual-inertial sensor configurations:

- Mono camera and IMU (Inertial Measurement Unit).
- Stereo cameras and IMU.
- Stereo cameras only.

To isolate the impact of rain on VO, we leverage VINS-Fusion in its stereo camera-only configuration running on the Intel RealSense D435i camera. This configuration excludes the IMU data, allowing us to focus solely on the performance of the visual odometry system under varying rain intensities.
To evaluate the performance of the VO system under varying rain intensities, we conducted controlled experiments within a high-entropy laboratory environment. The experiments encompassed two distinct navigation scenarios: static and moving.

- **Static Condition**: In this scenario, the VO system remains stationary on all axes, simulating a hovering drone. This configuration isolates the impact of rain on the camera data without additional movement influencing the VO estimation.
- **Moving Condition**: Here, the VO system follows a predefined rectangular trajectory of 140 × 160 cm with a constant height. This simulates a drone navigating at a desired altitude. The experiments are conducted by a trained user who moves the VO system over a stool with a constant velocity of approximately 0.2 m/s. This low velocity allows us to focus primarily on the rain's effect on the visual odometry system.

Rain Simulation

We employed a sprayer to simulate different rain conditions by varying its distance and inclination relative to the VO system [165]. Specific rain intensity levels are achieved based on the parameters detailed in Table 3.7. Additionally, we conducted control experiments without simulated rain to establish a baseline performance

reference for comparison.

To comprehensively analyze the impact of rain on the VO system,

Table 3.7: The rain conditions used during the experiments. "Dist." represents the distance between the sprayer and the box to simulate different rain intensities. To avoid the visual perturbation of the user on the camera scene, the inclination is w.r.t. the vertical axes in front/side of the camera.

Rain Conditions	Slanting	Vertical
Heavy	Dist. $<$ 10 cm Inclination $\sim 30°$	Dist. $<$ 10 cm Inclination $\sim 0°$
Medium	10 cm $<$ Dist. $<$ 20 cm Inclination $\sim 30°$	10 cm $<$ Dist. $<$ 20 cm Inclination $\sim 0°$
Low	Dist. $>$ 30 cm Inclination $\sim 30°$	Dist. $>$ 30 cm Inclination $\sim 0°$

we design two distinct rain simulation scenarios:

- **Slanting Rain**: This scenario, replicating the most common condition during medium/high-velocity drone navigation, simulates raindrops directly striking the camera lens. This configuration allows us to assess the immediate effects of raindrops on image quality and their influence on VO performance.
- **Vertical Rain**: This scenario simulates light rain or drizzle, often encountered during hovering or low-speed navigation. While vertical rain may not directly impact the camera lens with individual droplets, it can create a mist or film of water droplets on the lens surface. This scenario allows us to evaluate the performance degradation due to such a film obstructing the camera's view.

For each scenario presented above, we conduct experiments with three different rain intensities, namely, heavy, medium, and low (see Table 3.7). Moreover, to quantify the system's recovery time under adverse conditions, we measured the restoring time specifically in the slanting rain scenario (considered the most severe). This restoring time represents the duration required for the VO system to return to an acceptable performance level, with an average error

below 30 cm.
To ensure statistically robust results, all experiments are repeated 30 times under each condition. The rain simulation employed a constant water flow rate of 1.8 ml/s. This rate is chosen to mimic the natural spraying behaviour of a typical user, where a single press on a sprayer might result in a continuous stream equivalent to approximately 2.4 sprays per second.

3.2.4 DNN Development and Dataset

Dataset

To facilitate the development of robust rain classification systems for autonomous UAV navigation, we created a comprehensive dataset encompassing various rain conditions. This dataset is composed of raw colour images captured during the experiments described above. The dataset comprises seven distinct classes representing different rain conditions:

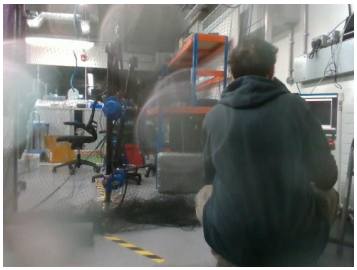
- Clear
- Slanting Heavy Rain
- Vertical Heavy Rain
- Slanting Medium Rain
- Vertical Medium Rain
- Slanting Low Rain
- Vertical Low Rain

Each class contains 48,000 images, resulting in a total of approximately 336,000 images. The dataset is divided into training (80%), validation (10%), and testing (10%) sets to facilitate the development and evaluation of machine learning models for rain classification. A representative selection of images from the dataset is presented in Figure 3.11.

DNN Training

To achieve accurate rain classification for autonomous UAV navigation, we employ three state-of-the-art deep learning architectures

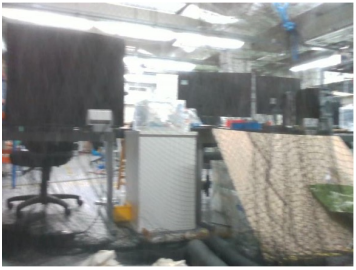
(a) Slanting heavy rain.



(b) Slanting medium rain.

(c) Slanting low rain.

(d) Vertical heavy rain.



(e) Vertical medium rain.

(f) Vertical low rain.

Figure 3.11: Example of dataset images.

because they present an excellent trade-off between computational complexity and classification performance [166, 167]:

- **MobileNetV2** ($\alpha = 0.35$) [2]: This architecture offers a well-balanced trade-off between performance and computational efficiency, making it suitable for resource-constrained UAV platforms. We employed a variant with an α parameter of 0.35 for this specific task.
- **MobileNetV3 small** ($\alpha = 0.35$) [168]: Similar to MobileNet V2, the MobileNetV3 small model prioritizes efficiency while maintaining good accuracy. We used a variant with an α parameter of 0.35 for optimal performance in this application.
- **SqueezeNet** [1]: This architecture is known for its compact design while achieving high classification accuracy.

All three DNNs are trained on a powerful NVIDIA GeForce RTX 4090 GPU using the following hyperparameters:

- Epochs: 100
- Image Input: $224 \times 224 \times 3$ (image size with width, height, and colour channels)
- Optimizer: Stochastic Gradient Descent (SGD)
- Batch Size: 64 (number of images processed per training step)
- Learning Rate Scheduler: Polynomial decay with an initial learning rate of 0.1, decaying to 0.001 over 10,000 steps using a square root (power of 0.5) decay function.

As shown in Table 3.8, all three DNN architectures (MobileNetV2, MobileNetV3 small, and SqueezeNet) exhibit a low memory footprint despite their potentially deep structures. This efficiency is attributed to the innovative building blocks incorporated into these architectures, which optimize the balance between model complexity and resource requirements.

3.2.5 Results

This section presents the findings from the experiments conducted using the setup described before. We employed two key metrics to

Table 3.8: Memory footprint and number of parameters of MobileNetV2, MobileNetV3 small, and SqueezeNet.

Architecture	Number of Parameters	Memory Footprint (MB)
MobileNet V2	419175	1.70
MobileNet V3 Small	336855	1.28
SqueezeNet	774503	2.95

quantify the impact of rain on the VO system's path estimation accuracy:

- **Standard Deviation**: This metric measures the variability of the estimated path compared to the actual (ground truth) trajectory. We will analyze the standard deviation for the static scenario. A higher standard deviation indicates a greater deviation from the true path due to rain's influence.
- **Root Mean Square Error (RMSE)**: This metric provides the average magnitude of the error between the estimated path and the ground truth, with the clear condition as the reference. Analyzing RMSE for the moving scenario will reveal how rain intensity affects the average error in path estimation.

Furthermore, we measure the restoring time specifically in the slanting rain scenario, considered the most severe. Restoring time refers to the duration required for the VO system to recover to an acceptable performance level, with an average error below 30 cm. This value indicates how long it takes for the system to regain sufficient accuracy after encountering heavy rain. To assess the effectiveness of the DNNs for rain classification, we use the following metrics based on the confusion matrix of the 7-class classifier:

- Average Accuracy: This metric represents the overall proportion of correctly classified rain condition samples across all classes. A higher average accuracy indicates the DNN's general ability to identify rain conditions accurately.
- Precision: This metric measures the fraction of correctly identified samples for each rain condition class. High precision for

a specific class signifies the DNN's ability to accurately distinguish that class from others.

- Recall: This metric indicates the proportion of true positives (correctly classified samples) for each rain condition class out of all actual positives in that class. High recall for a class suggests the DNN's effectiveness in capturing most of the relevant samples within that class.
- F1-Score: This metric combines precision and recall, providing a balanced view of the DNN's performance for each class. A high F1-score indicates a good balance between precision and recall for a specific class.

The DNN with the highest average accuracy, precision, recall, and F1-score across all rain classes will be identified as the best-performing model for rain classification.

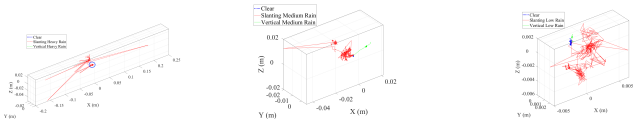
Static Condition

We first evaluate the VO system's performance under various rain conditions in a static scenario, where the system remained stationary throughout the experiment. Figure 3.12 compares the esti-

Table 3.9: Standard deviation computed for the experiments in static condition on the three axes x, y, and z."Slanting Heavy Rain" is the worst-case scenario (highlighted in red), while 'Vertical Low Rain' is the best-case scenario (highlighted in green).

Condition (Static)	**Error (σ_x, σ_y, σ_z)(mm)**
Clear (Reference)	0.05, 0.09, 0.2
Slanting Heavy Rain	10.5, 6.2, 7.9
Vertical Heavy Rain	0.1, 0.07, 0.3
Slanting Medium Rain	3.6, 5.1, 4.5
Vertical Medium Rain	3.5, 1.4, 0.4
Slanting Low Rain	1.0, 0.6, 1.4
Vertical Low Rain	0.09, 0.09, 0.2

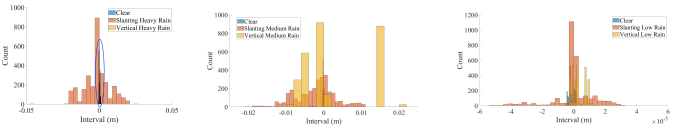
mated trajectories and data distribution across different rain conditions. Table 3.9 summarizes the standard deviation of the estimated position for each rain condition. Since the system is fixed at the origin (0, 0, 0) in all axes, the standard deviation reflects the



(a) Trajectory estimation of slanting and vertical heavy rain.

(b) Trajectory estimation of slanting and vertical medium rain.

(c) Trajectory estimation of slanting and vertical low rain.



(d) Data distribution of slanting and vertical heavy rain.

(e) Data distribution of slanting and vertical medium rain.

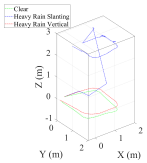
(f) Data distribution of slanting and vertical low rain.

Figure 3.12: Trajectory estimation and data distribution of the static scenario under the different rain conditions. In Figures "a" and "d", the clear and vertical rain trajectories and data distributions are highlighted by the blue circle.

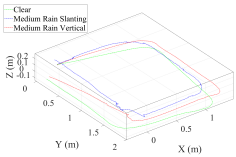
performance degradation relative to the ideal scenario (no rain). The results highlight the significant impact of slanting heavy rain, as shown in Figures 3.12a and 3.12d. The high standard deviation value in Table 3.9 confirms this observation, indicating a considerable drift in the estimated position. Conversely, the "Vertical Low Rain" scenario exhibits minimal error according to the standard deviation in Table 3.9, suggesting negligible performance degradation compared to the clear condition.

Moving Condition

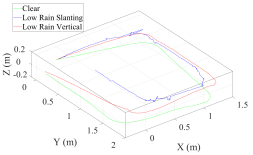
We further evaluate the VO system's performance under various rain conditions in a moving scenario. Here, the system follows a predefined rectangular trajectory at a constant height. Figure 3.13 compares the estimated trajectories under different rain intensity levels (heavy, medium, and low) within each rain direction (slanting and vertical). Table 3.10 summarizes the RMSE for each rain condition, calculated with the clear condition as the baseline. The



(a) Trajectory estimation of slanting and vertical heavy rain.



(b) Trajectory estimation of slanting and vertical medium rain.



(c) Trajectory estimation of slanting and vertical low rain.

Figure 3.13: Trajectory estimation and data distribution of the moving scenario under the different rain conditions.

Table 3.10: RMSE over the three axes and restoring time of the moving scenario. The worst-case scenario is "Slanting Heavy Rain" (highlighted in red), while the best-case scenarios are "Vertical Medium Rain" and "Vertical Low Rain" (highlighted in green).

Condition (Moving)	RMSE (x, y, z)(m)	Restoring Time (s)
Slanting Heavy Rain	1.3, 0.9, 2.5	32.9
Vertical Heavy Rain	0.4, 0.4, 0.09	NA
Slanting Medium Rain	0.5, 0.5, 0.3	20.1
Vertical Medium Rain	0.3, 0.3, 0.08	NA
Slanting Low Rain	0.8, 0.9, 0.4	13.34
Vertical Low Rain	0.3, 0.4, 0.07	NA

results reinforce the significant impact of slanting heavy rain. As shown in Figure 3.13a, this scenario introduces a substantial and unacceptable drift in all axes, particularly evident in the vertical direction. Table 3.10 confirms this observation with a high RMSE value for the slanting heavy rain scenario. In contrast, the vertical rain scenarios (both low and medium) exhibit considerably less se-

vere errors. Their estimated trajectories in Figures 3.13b and 3.13c show drift in the order of tens of centimetres, and the corresponding RMSE values in Table 3.10 are lower compared to the slanting rain condition.

DNN Results

We evaluate the performance of the three DNN architectures (MobileNetV2, MobileNetV3 small, and SqueezeNet) using their respective confusion matrices (refer to Tables 3.11, 3.12, and 3.13 for detailed results). The evaluation metrics include precision, recall, and F1-score for each of the seven rain classes, computed on the test set. Overall, all three DNNs achieved good performance across most rain classes, particularly for "Clear" and "Slanting Heavy Rain". These two classes represent the extremes of the rain conditions studied, making them easier for the DNNs to distinguish. However, a common challenge across all DNNs is observed in the "Vertical Low Rain" class, where recall is the lowest (highlighted in red in the tables). This indicates a higher number of false negatives, meaning the DNNs sometimes misclassified light vertical rain as clear weather. This behaviour is understandable as the visual influence of light vertical rain is minimal and can be very similar to clear conditions, as confirmed by the VO system analysis. In the context of autonomous UAV navigation, a false negative for "Vertical Low Rain" is not critical. The slight visual perturbation caused by light vertical rain is unlikely to significantly impact navigation safety.

Table 3.11: MobileNetV2 performance of each class. The worst result consists of the recall in the "Vertical Low Rain" scenario (highlighted in red).

MobileNet V2	Precision	Recall	F1-score
Clear	0.99	0.99	0.99
Slanting Heavy Rain	0.99	0.98	0.99
Vertical Heavy Rain	0.78	0.99	0.88
Slanting Medium Rain	0.97	0.79	0.87
Vertical Medium Rain	0.76	0.99	0.86
Slanting Low Rain	0.96	0.91	0.93
Vertical Low Rain	0.93	0.68	0.79

Table 3.12: MobileNetV3 Small performance of each class. The worst result consists of the recall in the "Vertical Low Rain" scenario (highlighted in red).

MobileNet V3 Small	Precision	Recall	F1-score
Clear	0.97	0.99	0.98
Slanting Heavy Rain	0.98	0.99	0.98
Vertical Heavy Rain	0.86	0.95	0.90
Slanting Medium Rain	0.97	0.88	0.92
Vertical Medium Rain	0.75	0.97	0.84
Slanting Low Rain	0.93	0.91	0.92
Vertical Low Rain	0.89	0.62	0.73

Table 3.13: SqueezeNet performance of each class. The worst result is the recall in the "Vertical Low Rain" scenario (highlighted in red).

SqueezeNet	Precision	Recall	F1-score
Clear	0.97	0.99	0.98
Slanting Heavy Rain	0.98	1.00	0.99
Vertical Heavy Rain	0.95	1.00	0.97
Slanting Medium Rain	0.94	0.82	0.88
Vertical Medium Rain	0.76	0.98	0.86
Slanting Low Rain	0.98	0.88	0.93
Vertical Low Rain	0.83	0.72	0.77

Table 3.14 summarizes the overall accuracy, precision, recall, and F1-score for all three DNNs. The results indicate comparable performance across the architectures, with all achieving metrics above 90%. This suggests their effectiveness in rain classification for UAV navigation.

Table 3.14: Results on the test dataset of the three architectures.

Architecture	Accuracy	Precision	Recall	F1-score
MobileNet V2	0.90	0.91	0.91	0.90
MobileNet V3 Small	0.90	0.91	0.90	0.90
SqueezeNet	0.91	0.92	0.91	0.91

While all three DNNs perform well, MobileNetV3 Small offers a compelling advantage for resource-constrained UAV platforms. As shown in Table 3.8, it boasts the lowest memory footprint among the tested architectures. Additionally, Table 3.15 reveals that MobileNetV3 Small has the fastest classification latency of approximately 10 milliseconds. This combination of low memory usage and fast inference speed makes MobileNetV3 Small the most suitable choice for deploying a stable, real-time rain classification solution on small UAVs with limited on-board computational resources.

Table 3.15: Execution time in seconds of a classification cycle of the three classifiers on the Intel NUC 11.

Classifiers	**Avg. (s)**	**Var.**$(\times 10^{-5})$ (s^2)	**Max. (s)**	**Min. (s)**
MobileNet V2	0.0706	3.2123	0.0944	0.0651
MobileNet V3 Small	0.0107	0.2062	0.0167	0.0087
SqueezeNet	0.5097	73.476	0.6385	0.4791

3.2.6 Conclusions

The growing prevalence of UAVs necessitates addressing challenges that hinder autonomous navigation in various weather conditions. While visual odometry (VO) systems are widely used, their behaviour under adverse weather like rain remains under-explored in the literature. This gap necessitates research to ensure safe and reliable drone flight operations.

This study investigates the impact of rain on VO system performance for UAV navigation. Our analysis identifies slanting heavy rain as the most critical scenario, introducing significant path estimation errors (1-2.5 meters) that render navigation unsafe. Other "slanting rain" scenarios also exhibit higher errors compared to vertical rain, but their acceptability depends on specific application tolerances. To mitigate rain's influence and enhance environmental awareness, we propose a deep learning-based approach. We trained and compared three DNN architectures, achieving an accuracy of around 90%, to classify colour images captured by the VO system into seven rain conditions: clear, varying degrees of slanting and ver-

tical rain (heavy, medium, low). The best-performing DNN achieved a frame rate of 97 FPS, enabling real-time classification.
This classification information can be used to suggest appropriate actions for the drone based on the prevailing weather. Potential actions include switching to alternative navigation systems, adjusting flight paths, or even landing. This approach has the potential to significantly reduce the risk of accidents and enhance drone reliability and operational flexibility. However, it is important to acknowledge the potential limitations of the DNN-based approach. The accuracy of 90% might not be perfect in all real-world scenarios, and factors like variations in rain characteristics or sensor quality could affect performance.

Chapter 4

Industrial Visual Inspection

The quality inspection of industrial products is a fundamental step in large-scale production as it boosts the yield and reduces costs. Intelligent embedded platforms with built-in tiny machine learning algorithms and cameras can automate quality inspection. However, running complex deep learning algorithms in low-cost and low-power embedded devices is still challenging because of the limited memory and energy resources. This chapter [1] presents an innovative sensor system with three MCU-based tinyML cameras capable of automatic artefact and anomaly detection in plastic components. The system consists of a top camera for identifying shape defects and two side cameras for colour anomalies. Data processing is executed locally with the tinyML reducing the data transmission to a few bytes. Two state-of-the-art convolutional neural networks (CNN) are evaluated, namely MobileNetV2 and SqueezeNet. Results show

[1] The work presented in this chapter has been published in the following papers:

- Albanese, A., Nardello, M., Fiacco, G., and Brunelli, D. (2022). Tiny machine learning for high accuracy product quality inspection. IEEE Sensors Journal, 23(2), 1575-1583.
- Albanese, A., and Brunelli, D. (2023). Industrial Visual Inspection with TinyML for High-Performance Quality Control. IEEE Instrumentation and Measurement Magazine, 26(8), 17-22.

how both architectures - with appropriate compression techniques – are suitable to be evaluated by resource-constrained microcontrollers. The networks achieve 99% classification accuracy while maintaining suitable real-time performance, respectively equal to 5 FPS and 2 FPS.

4.1 Introduction

The Industrial Internet of Things (I-IoT) is becoming essential for companies to optimize their production processes. Adding distributed intelligence along the production lines can lead to consistent cost savings. Machine learning can be a crucial tool for supervising products and production processes [169]. It is possible to generate models that can quickly inspect the quality of a product (e.g., the presence or absence of a correct label on a bottle or the classification of defects on the surface) with statistical methods and databases, thus avoiding possible waste in production lines [170]. So far, cloud-based architectures have been used for many industrial inspection applications by exploiting servers with unlimited computing resources to carry out complex data processing. The cloud computing approach usually employs high-resolution industrial cameras providing high-quality images for accurate analysis. Even though they provide superior service quality, they are expensive solutions in terms of cost and performance [171]. Furthermore, this approach is highly conditioned by the network quality, which can introduce a significant latency or lead to service degradation if the connection is lost. Also, scalability is affected as more sensors pressure the cloud infrastructure (i.e., bandwidth and data storage requirements).
A solution can be represented by low-cost intelligent electronic devices, as such embedded platforms can guarantee a reduced development cost without affecting the quality of service [172]. Using the computational capacity that embedded systems have achieved, it is possible to design complex visual inspection systems – powered by deep learning algorithms – directly on intelligent sensors and actuators, using the so-called tinyML approach. TinyML expands the *edge computing* paradigm bringing complex data processing closer to the data source. This approach improves the application's responsiveness and efficiency, reducing the amount of data

transmitted [173]. Thus, cloud computing limitations associated with data throughput and costs are avoided. In-pixel processing architectures [174] could be used shortly to reduce the data exchange between the sensor and the nearby microcontroller. Also, event cameras can improve the system's efficiency by transmitting only detected conditions in the image (i.e., events in a scene). However, these solutions are still under development and unavailable on the market.

This chapter presents the design, implementation, and evaluation of an automatic visual inspection I-IoT system based on image data processing with tinyML, designed for large-scale product quality inspection. We addressed the specific requirements of a company leader in producing plastic parts through the injection moulding process. They provided the dataset used to develop the deep learning models, the specifications about the responsiveness of the visual inspection system, and the list of possible anomalies during the process. For this reason, this work consists of a unique scenario with real case problems. The system is currently used in a dozen of the company's machines for long-time and large-scale testing. Two convolutional neural networks (CNNs) for image classification are trained, tested, and compared, namely MobileNetV2 [2] and SqueezeNet [1]. Then, the CNN models are compressed and deployed in the target platform, namely OpenMV Cam H7 Plus, for image processing and neural classification directly on the microcontroller unit (MCU). Results highlight the perfect fit for this use case due to their optimized structure for resource-constrained environments.

In particular, the main contributions of this chapter are:

- The design of a cyber-physical system for product quality inspection, capable of detecting the defects of plastic moulded objects (Figure 4.1).
- The design, optimization, and deployment of tiny neural networks (NNs) for object classification on resource-constrained cameras.
- The creation of a custom dataset used to train and test different NN architectures.
- The evaluation and comparison of performances between the two tiny NNs, namely MobileNetV2 and SqueezeNet.

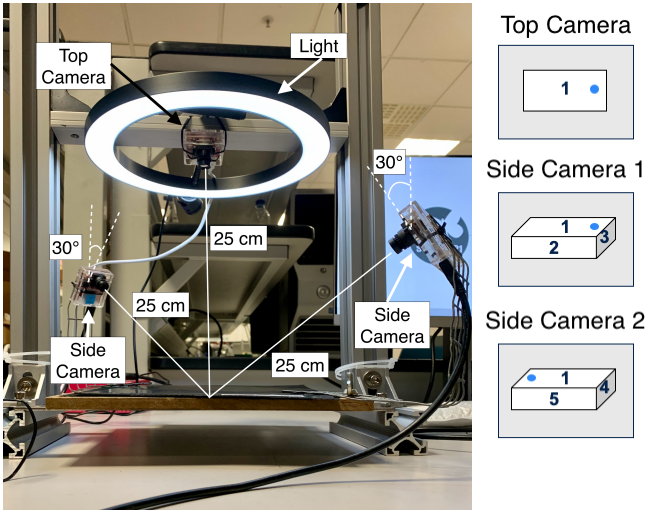


Figure 4.1: Picture of the system's setup. The top camera is in the centre of the ring light. Each camera is located at a distance of 25 cm from the working plane to ensure in-focus objects. Side cameras have an orientation angle of 30° with respect to the vertical axes.

- The system characterization by examining its execution time and energy consumption during image pre-processing and classification.

4.2 Related Work

Edge devices with ML capabilities are recently gaining interest in designing intelligent IoT infrastructures that limit data exchange to a few bytes. AI edge processing focuses on moving the inference part of the AI workflow on the device by keeping data locally to improve latency and bandwidth [175].

4.2.1 Industrial quality inspection

The main challenges in using ML algorithms in industrial manufacturing environments are the limited processing capabilities, the

use of big data, memory and energy constraints, and, sometimes, real-time processing [176]. Recent technologies for industrial visual inspection are based on line-scan cameras, spectrometers, or high-resolution cameras. These systems are expensive and require a significant amount of time to inspect one piece [177]. However, companies that want to offer high-quality products and optimize their production processes or costs need comprehensive and reliable quality inspection tools. In [178], a framework for the detection of glass bottle bottom defects is implemented using a combination of the visual attention model and wavelet transformation. The system achieves a recall of around 92%, requiring only 535 ms of computational time on a low-performance laptop CPU. The possible quality improvements in industrial processes are also highlighted in [179]. By exploiting a machine learning vision-based classification model, the authors highlight how the histogram-based droplet detection and micrograph classification approach can be exploited to determine when the emulsification process is completed automatically.
However, inspection systems are usually deployed in unreachable positions, making maintenance difficult. Thus, such systems must be standalone and used with the "deploy and forget" approach. Also, energy resources play a fundamental role in ensuring the system's reliability. For those reasons, research in machine learning has increasingly shifted to move data evaluation where it is generated, reducing and optimizing the used resources. In [176], the authors propose a quality inspection system that uses supervised ML algorithms in edge devices. This work supports manufacturing companies with a predictive model-based quality inspection system to predict the final product quality based on the recorded parameter of the process.
On the other hand, deep learning methods are data-hungry: they need a large amount of annotated data which is labor-intensive and time-consuming. As a workaround to these limitations, the authors in [180] have proposed a segmentation-aggregation framework to train object detectors from annotated visual data for automatic industrial visual inspection. They have limited the data annotation to label the image class and avoid the expensive task of the bounding box coordinate annotation. For this purpose, developing an accurate dataset for ML training is crucial to obtaining a precise and efficient system.

4.2.2 Deep learning architecture and optimization techniques

In the last few years, fostered by new-generation microcontrollers, multiple neural network architectures were developed for resource-constrained devices [130]. Thanks to the study of innovative pruning and quantization techniques [120], it is possible to drastically reduce the ML model complexity while maintaining the same prediction accuracy. By combining those traditional techniques with more recent Neural Architecture Search (NAS) [181], and Federated Learning [182] approaches, it is possible to deploy AI-based systems in MCUs with impressive low energy consumption and high accuracy [183]. In the literature, we can already find multiple implementations, like MobileNetV2 [2] and SqueezeNet [1] and, more recently, new cutting-edge deep architecture, namely MobileNetV3 small [168], EfficientNet [184], and MCU Net V1 and V2 [183, 185]. In our proposed implementation, we have preferred to stick with the well-investigated MobileNetV2 and SqueezeNet as the preliminary results obtained satisfied the requirements of this application. We have thus preferred focusing on the deployment and long-term evaluation of the whole system in a real industrial large-scale test to assess the real performance of the system. This has allowed us to optimize the selected network further and achieve the results presented in Section 4.7.

4.3 System Architecture

The architecture of the visual inspection system is shown in Figure 4.2. The workflow consists of positioning the produced items on a conveyor belt and moving them until they reach three cameras that acquire pictures of the items from different perspectives. A ToF (Time of Flight) sensor is placed on the side of the belt and alerts the cloud gateway that the object has arrived at the exact location. The gateway stops the belt, and the image classification phase begins. By using DNNs and computer vision algorithms, the MCU on the camera classifies objects, like those in Figure 4.3, as conformant, with shape problems, or with colour anomalies. Inferences and detection results are sent to the gateway, which controls and rejects the part if it does not meet the requirements. In detail, this process can be divided into four steps.

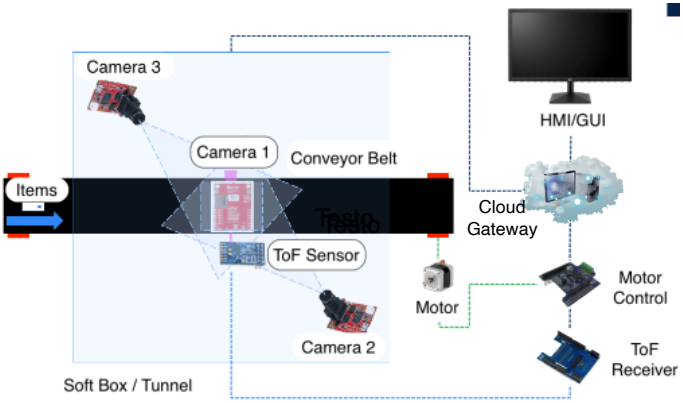


Figure 4.2: System architecture. It consists of a conveyor belt, a ToF sensor that detects the presence of an object, three MCU-based cameras responsible for image processing and classification, and the cloud gateway which retrieves or rejects the piece according to the classification results.

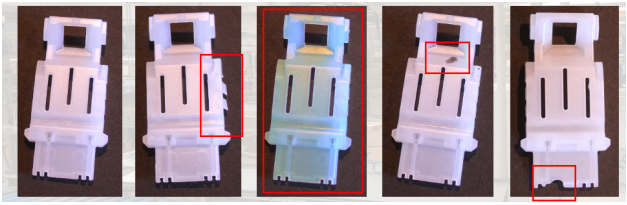


Figure 4.3: Possible defects of the objects. From left: conforming object, deformed object, polluted object, object with stains, and incomplete object.

4.3.1 Item Upload

The first phase consists of sliding the object on the conveyor belt. The rotation of the object during the feeding has no constraints, so the components can assume a random rotation. The conveyor belt colour was carefully chosen as “matte black” to avoid unwanted light reflections that can generate image distortions and lead to wrong classifications.

4.3.2 Item Movement

The object slides on the conveyor belt, which is moved by a stepper motor positioned at the roller. The motor is controlled by the M4 core of the STM32MP1 MPU through the X-NUCLEO-IHM03M1 expansion board. Moreover, an X-NUCLEO-6180XA1 expansion board detects the presence of the items on the belt through a ToF sensor. The X-NUCLEO-6180XA1 also measures the ambient light, used to derive the light conditions when tagging and classifying items. Both expansion boards are connected to the cloud gateway.

4.3.3 Object Classification

Object classification is carried out with an edge computing approach by the arrangement of three OpenMV H7 Plus cameras, as shown in Figure 4.1. Two different models are used for the classification of shape and colour anomalies. The position and the number of the cameras were chosen based on the types of defects to detect. Anomalies mainly occur only on 5 of the 6 faces of the object. This means that the field of view (FoV) of a single camera cannot cover all the object's faces. The three cameras are placed in three different locations to ensure that all faces are within the cameras' FoV. A "top camera" is positioned orthogonally to the belt and focuses on the 2D plane of the object. The other two cameras, called "side cameras", are placed in a specular position with an orientation angle of 30° concerning the vertical axes. They are placed on either side of the object and expand the field of view on the remaining four faces. The distance of the lens from the object is 25 cm to ensure focus objects. Moreover, the 20-degree FoV lens guarantees the inclusion of an item within a picture with some extra space around it to compensate for possible delays when stopping the belt. The three cameras take three different images of the component on the belt. These images are the input of DNNs deployed on the MCU responsible for classifying shape and colour imperfections. The "shape-defect" anomalies occur on the perimeter of the object. It follows that the top camera is best suited to select this type of nonconformity. The side cameras are used to detect color-defected objects.

4.3.4 Post-processing

The cloud gateway, according to the value obtained from the three cameras, enables the conveyor belt motor when the objects are conformant. On the contrary, if the result of the prediction is a non-conforming object, it proceeds to eliminate the component by activating a plunger.

4.4 IoT Device for Inspection and Monitoring

Advanced visual inspection and monitoring need cutting-edge standalone IoT devices to minimize maintenance and costs, by improving its deployment in industrial plants. This device [2] can be powered by a battery and it is equipped with LTE connectivity to use the NB-IoT standard, making it deployable in almost any environment where a power source or a stable Wi-Fi connection is not available. Furthermore, it can monitor the environmental conditions ensuring a continuous streaming of data displayed in a dedicated user interface. The device prototype is shown in Figure 4.4. It

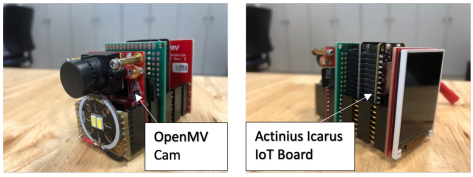


Figure 4.4: Prototype of the industrial monitoring device composed of an MCU-based camera and an IoT board with LTE connectivity.

is composed of two active parts, namely the OpenMV Cam (i.e., an MCU-based camera with tinyML capabilities), and the Actinius Icarus IoT board based on the Nordic nRF9160 SiP with LTE-M and GNSS modems. They are interconnected as shown in Figure 4.5. In particular, the camera is responsible for the user interface and the

[2] This device has been presented in a patent application developed in collaboration with "Telecom Italia":

- Albanese, A., Barchi, F., Brunelli, D., Elia, N., and Gotta, D. (2023). Method and System for Controlling a Shipment.

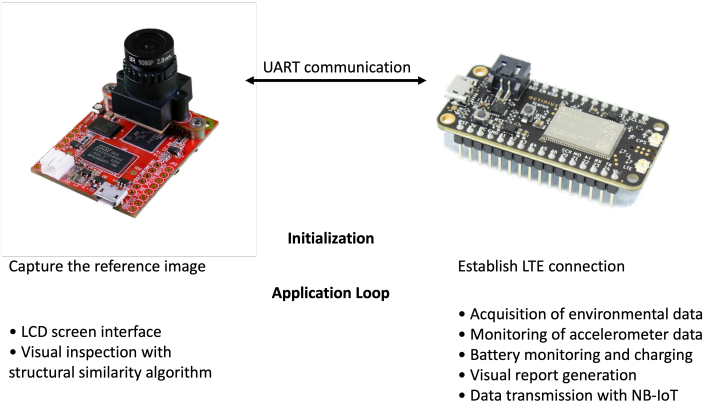


Figure 4.5: Functionalities of the inspection and monitoring device.

ML processing, while the Actinius Icarus IoT board is responsible for the LTE connectivity and data transmission via NB-IoT and environmental monitoring. The proof of concept of the monitoring IoT device has enormous potential as it is independent, battery-powered, running ML on the edge. It can be employed in different applications, from industrial monitoring to advanced asset tracking.

4.5 Tiny Neural Networks

4.5.1 Network Architectures

Most DNN architectures require high computational capacity, focusing the deployment on specific high-performance computational units. In this application, the available resources are limited because the target board is an MCU. This led to a challenge in researching and optimizing DNNs. Two of the best-performing DNNs specifically designed for embedded systems are chosen, namely MobileNetV2 [2] and SqueezeNet [1]. MobileNetV2 topology is defined in Table 4.1, while the main block of SqueezeNet is shown in Figure 4.6 (i.e., the “Fire Module”).

Table 4.1: Topology of MobileNetV2. "n" denotes the replicas of the same layer, "c" the number of output channels, "s" the stride, and "t" the expansion factor [2].

Input	Operator	t	c	n	s
$224 \times 224 \times 3$	*conv2d*	-	32	1	2
$112 \times 112 \times 32$	*bottleneck*	1	16	1	1
$112 \times 112 \times 16$	*bottleneck*	6	24	2	2
$56 \times 56 \times 24$	*bottleneck*	6	32	3	2
$28 \times 28 \times 32$	*bottleneck*	6	64	4	2
$14 \times 14 \times 64$	*bottleneck*	6	96	3	1
$14 \times 14 \times 96$	*bottleneck*	6	160	3	2
$7 \times 7 \times 160$	*bottleneck*	6	320	1	1
$7 \times 7 \times 320$	$conv2d 1 \times 1$	-	1280	1	1
$7 \times 7 \times 1280$	$avgpool 7 \times 7$	-	-	1	-
$1 \times 1 \times 1280$	$conv2d 1 \times 1$	-	k	-	-

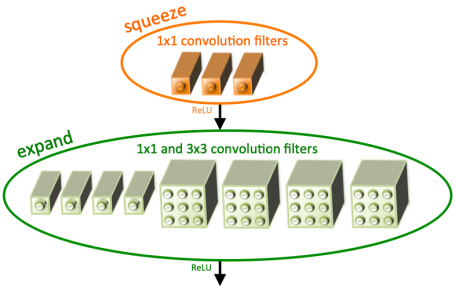


Figure 4.6: SqueezeNet micro-architectural view of convolution filters and fire module [1].

4.5.2 Model Compression

Model compression is a fundamental step to deploying deep learning models in resource-constrained devices. In this application, three compression techniques are used to make possible the deployment of MobileNetV2 and SqueezeNet on the OpenMV Cam H7 plus. Even though two datasets are used to compose the visual inspection system for colour and shape anomalies, they share the same model architecture and input image size; therefore, the number of

parameters and the required resources are the same.

Parameter minimization

DNN's parameters are the sum of the weights and biases of each layer. Convolution or fully connected layers present a high number of parameters. Therefore, the number of parameters is proportional to the number of convolution layers and affects the final model dimensions. It is possible to reduce the number of layers to obtain a lighter model. However, this operation can reduce the overall accuracy, especially if the network becomes too shallow. Starting from this idea, MobileNetV2's blocks are reduced from 17 to 14, and the fire modules of SqueezeNet are reduced from 8 to 5. The effect of parameter optimization is presented in the last two rows of Table 4.4.

Pruning

After training the DNN models with the optimized architectures, pruning is applied to optimize the model complexity further. It permits cutting off weights irrelevant for prediction purposes (e.g., weights close to zero). We used the "Polynomial Decay" method to apply sparsity to the DNN. This method uses a range of sparsity values to mask weights, starting from those with less significant values and increasing them until the final sparsity is reached. In this case, both DNNs were pruned with a sparsity range of [20, 50] expressed as a percentage of removed weights, obtaining a model composed of 50% of the original parameters. This threshold is chosen to not lose accuracy. The result of the pruning operation is shown in Table 4.2.

Table 4.2: Number of parameters before pruning and number of non-zero parameters (NNZ) after pruning.

	Parameters before pruning	**Sparsity**	**NNZ after pruning**
MobileNetV2	234 914	50%	117 457
SqueezeNet	120 930	50%	60 456

Quantization

Quantization is essential to deploy DL models in MCU and improve hardware acceleration latency and power efficiency. A model quantized with 8-bit representation results in a model 4x smaller and 1.5x-4x faster in computations. In this work, weights, activations, and network inputs and outputs are quantized. As a result, a "full-integer" model is obtained.

4.6 System Implementation

4.6.1 Image Pre-processing

The pre-processing algorithm analyzes the images in the camera MCU to highlight the relevant features useful for classification purposes. In this use case, the object arrives on a dark belt; thus, algorithms for background removal are used to avoid light reflection problems. Moreover, the object to classify may not be placed exactly in the centre of the acquisition window. This means that clipping can cause the elimination of fundamental data for classification. For this reason, the developed algorithm considers the component's position within the image window, therefore avoiding incorrect clipping. Three well-known computer vision algorithms are used: the "Canny algorithm", "Blob detection", and "Otsu's method". The pre-processing algorithm works as follows:

- Capture an image and create a copy.
- Conversion of the image from RGB565 to gray-scale.
- Canny algorithm to find the component contour.
- Search for blobs inside the Canny image.
- Blobs merging.
- Blob center computation.
- Check the position of the centre to avoid wrong cropping.
- Image crop by taking as reference the obtained centre.

The images are acquired by setting the sensor size to QVGA resolution (i.e., 320×240). After the Canny algorithm, we binary the image in the range [0,1] in which the pixels with value 1 are the contours of the plastic component. The Blob detection algorithm obtains different blobs of 1-pixel size referred to each pixel with value 1. The blob fusion represents the four corners of the image, taking into account their outermost positions. Once a single blob is obtained, its centre is calculated as a reference point for the crop to avoid clipping. Then, images are resized into 160×160 size, and the background is removed. The resulting images include only the object and straighten every other pixel regarding the background as a value of 0 in the RGB range. At this point, Otsu's method is used to find an optimal threshold to execute background removal.

4.6.2 Dataset Acquisition

The collection of a robust dataset is a key step in DL algorithm development. The three-camera arrangement uses two different DNN models, thus, it is necessary to collect two separate datasets: one for the top camera and one for the side cameras. The two datasets are composed of images pre-processed with the algorithm presented in Section 4.6.1.

(a) (b) (c)

Figure 4.7: (a) pre-processing on the top camera showing a "conformant" component. (b) pre-processing done on the side camera showing a "color defected" component. (c) pre-processing done on the side camera showing a "conformant" component.

Figure 4.7 shows the results obtained by the pre-processing algorithms in the top camera and side cameras, respectively. In particular, Figure 4.7a is related to the pre-processing result from the top camera of a "conformant" component. Figure 4.7b shows a "colour

defected" component obtained after pre-processing. In this case, the background pixels and the stained part of the component have a value of 0 in the RGB channels. The rest of the image is in the RGB colour space, where each pixel can take any value in the range [0,255]. Figure 4.7c shows a "conformant" component; therefore, despite the previous case, no pixel value about the component is set to 0, but only the background pixels.
The pre-processed images are sent to the cloud gateway responsible for collecting the dataset. In this way, the camera does not keep images in memory but sends them to the gateway through the Remote Procedure Call (RPC) library embedded in Micro-Python. This allows the camera to connect to another device and execute remote procedure calls on the camera. The complete dataset is acquired with the setup shown in Figure 4.1. Each component is placed within the field of view of the three cameras, with different orientations and a minimum of random rotation to extend the heterogeneity of the dataset. Then, through a GUI, each image is tagged by selecting the class of the object.
More than 500 images are acquired for each camera in a balanced manner (i.e., each class includes the same number of images). Moreover, data augmentation is performed to increase the dataset size. Rotation and translation are used as image transformations for better generalizing the DNN's inputs. This operation is necessary to increase the number of images and replicate the real scenario where components can be placed in different positions and with minimal rotations in the camera field of view. The dataset is augmented by rotating the image in a range of [10, -10] degrees. However, this process is only done concerning the datum of the top camera because the two side cameras capture an image of the object laterally, therefore in a perspective way. A rotation of the image can cause distortion, thus, a wrong dataset optimization.
Given that the component can assume various positions, also image translation is used as image transformation. Considering that we are using square images with a rectangular object, the translation along the image width is set to 4% of the total (i.e., a random translation in the range [-9, 9] pixels). The translation in amplitude is 10% of the total (i.e., a random translation in the range [-22, 22] pixels along the height). The dataset size is summarized in Table 4.3 and is organized in sub-folders to simplify the label extraction.

Table 4.3: Overview of the dataset size for the top camera and the side cameras after data augmentation.

160x160px	**Top Camera**		**Side Camera**	
	Conformant	Shape Defected	Conformant	Colour Defected
Train	3709	4068	2500	2409
Validation	1045	1145	691	672
Test	863		544	
Total	10830		6816	

4.6.3 Training

The training of MobileNetV2 (alpha parameter equal to 0.35) and SqueezeNet is carried out with the parameters shown in Table 4.4. Model weights are initialized with random values. The number of epochs is set by using the *early stopping callback by validation accuracy* approach. This technique permits stopping the training session when the validation accuracy is below a certain threshold (i.e., 99.5%) to be less prone to network overfitting. The model evaluation is presented in Section 4.7.1. The resources needed for the deployment in the OpenMV Cam H7 Plus are summarized in Table 4.5.

Table 4.4: NN training parameters.

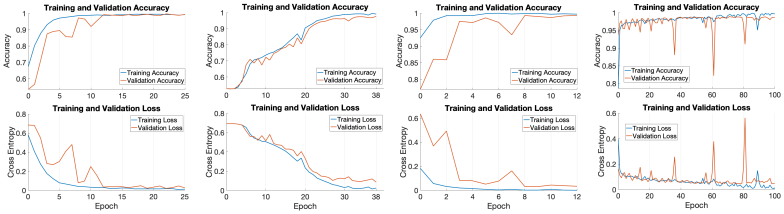
	Network Architecture	
	SqueezeNet	**MobileNetV2**
Batch size	32	32
Initial learning rate	10^{-5}	10^{-5}
Input image	RGB 160×160	RGB 160×160
Optimizer	Adam	Adam
Loss function	Binary cross-entropy	Binary cross-entropy
Source framework	Tensorflow	Tensorflow
♯ of parameters base model	723 522	412 770
♯ of parameters modified model	337 090	176 386

Table 4.5: Resources needed by the developed models to perform inference on OpenMV Cam H7 Plus.

160x160px	**Float32 model** (opt. parameters) Flash (KB)	RAM (KB)	**Optimized model** (pruning, quantization) Flash (KB)	RAM (KB)	**Compress Factor**
MobileNetV2	635.29	1490	172.26	381.22	3.8 ×
SqueezeNet	1290	1780	334.35	455.25	3.9 ×

4.7 Results and Evaluation

4.7.1 Tiny Neural Network Evaluation



(a) Training and validation accuracy of the MobileNetV2 architecture for the top camera.

(b) Training and validation accuracy of the SqueezeNet architecture for the top camera.

(c) Training and validation accuracy of the MobileNetV2 architecture for the side camera.

(d) Training and validation accuracy of the SqueezeNet architecture for the side camera.

Figure 4.8: Training and validation accuracy for both networks and cameras.

Top Camera

Figures 4.8a and 4.8b show the training results and validation accuracy of both architectures for the top camera dataset. To reach the desired validation accuracy value, MobileNetV2 and SqueezeNet need 25 and 38 epochs, respectively.

Side Cameras

Figures 4.8c and 4.8d show the results of the training session for the side camera dataset evaluated with MobileNetV2 and SqueezeNet, respectively. In this case, MobileNetV2 needs only 12 epochs to reach the desired validation accuracy, while SqueezeNet does not reach the set validation accuracy and uses the maximum number of epochs to complete the train (i.e., 100). Figure 4.8c shows that the validation loss value is low from the first epoch, which implies a rapid growth of the training accuracy. On the other hand, the plot in Figure 4.8d shows different negative peaks in the validation accuracy due to the wrong weight update during backpropagation. In this case, parameter optimization has influenced the performance of SqueezeNet.

4.7.2 Top Camera Test

The top camera aims to classify "conformant" and "shape-defected" objects. MobileNetV2 and SqueezeNet are evaluated by considering accuracy, precision, recall, and f-score, as well as the loss in accuracy throughout the optimization operations. The test is conducted with 863 images, where 412 of them are "conformant" and 451 are "shape-defected" images. Their results are summarized in Table 4.6. Even though the models are highly compressed, the loss in performance is negligible.
Figures 4.9a and 4.9b show the Grad-CAM heatmap of MobileNetV2 and SqueezeNet, respectively. Grad-CAM is a tool to reveal zones where the network extracts features for classification. Here, the yellow circle highlights the shape anomalies (i.e., a missing pin), while the red circle highlights the region where most of the features are extracted by the deep learning model. In this case, the DL model extracts most of the features where the imperfection is located to produce the classification result. The MobileNetV2 heatmap in Figure 4.9a shows that only the image portion that includes the object, especially the right side, is used for feature extraction. This leads to a more generalized CNN, which processes almost the whole object to classify shape defects. On the other hand, the SqueezeNet heatmap in Figure 4.9b highlights only the image part related to the anomaly (upper right corner). It means that the model is specialized for this type of imperfection and cannot generalize as MobileNetV2 does. Furthermore, it is crucial to minimize false negatives (FNs) in an

industrial visual inspection system to avoid missing detection of defects. As shown in Table 4.6, MobileNetV2 outperforms SqueezeNet achieving a recall of 100% (i.e., any FN is predicted during the test).

Table 4.6: Comparison of MobileNetV2 and SqueezeNet performance for the top camera. "Float 32" refers to the model with optimized parameters, while "Optimized" refers to compressed models with pruning and quantization.

160x160px	**MobileNetV2**			**SqueezeNet**		
	Float32	Optimized	Δ	Float32	Optimized	Δ
Accuracy	99.5%	98.9%	0.6%	98.6%	98.4%	0.2%
Precision	99%	98%	1%	99%	99%	0
Recall	100%	100%	0	98%	98%	0
F-score	99.5%	99%	0.5%	98%	98%	0

4.7.3 Side Cameras Test

Side cameras classify "conformant" and "color-defect" objects. The same evaluation for the top camera is conducted with 544 images. Table 4.7 summarizes the results. In this case, the comparison reveals good performance also with a different dataset, and the loss in accuracy due to the optimization process is negligible. Figures 4.9c and 4.9d show the Grad-CAM heatmap of a color-defected component of MobileNetV2 and SqueezeNet, respectively. Here, the yellow circle highlights the colour anomalies (i.e., a stain), while the red circle highlights the region where the deep learning model extracts most of the features. It is visible that feature extraction involves the portion of the image where the defect is located. However, Figure 4.9d confirms the worsening of the performance of SqueezeNet. It highlights that most of the features useful for classification are extracted from the region in the upper-left corner, while features in the bottom-right corner are less useful for classification purposes. This phenomenon does not occur in the MobileNetV2 case (Figure 4.9c), where the whole region that covers the object is used to extract features to classify colour anomalies. It means that MobileNetV2 can better generalize colour anomalies than SqueezeNet. Furthermore, MobileNetV2 outperforms Squeezenet by analyzing the FNs, achieving a recall of 99% after the optimization process (i.e., only 1% of the samples are classified as FN during the test).

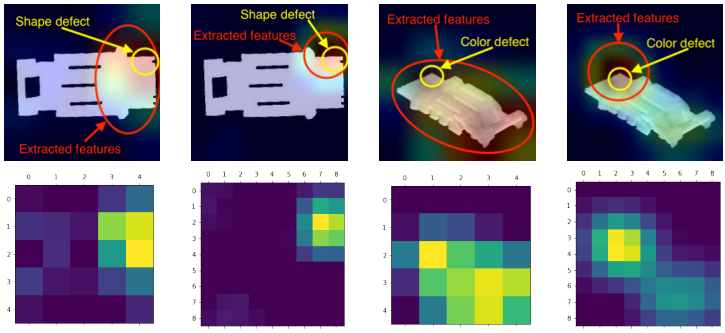


Figure 4.9: Grad-CAM heatmap of MobileNetV2 and SqueezeNet for the top camera ((a), and (b)), and side camera ((c), and (d)), respectively.

Table 4.7: Comparison of MobileNetV2 and SqueezeNet performance for the side cameras. "Float 32" refers to a model with optimized parameters, while "Optimized" refers to compressed models with pruning and quantization.

160x160px	**MobileNetV2**			**SqueezeNet**		
	Float32	Optimized	Δ	Float32	Optimized	Δ
Accuracy	100%	99.6%	0.6%	98.4%	98.2%	0.2%
Precision	100%	100%	0	100%	100%	0
Recall	100%	99%	1%	96%	96%	0
F-score	100%	99.5%	0.5%	98%	98%	0

4.7.4 Inference on Cameras

The developed NN models are deployed in the setup shown in Figure 4.1. Their performance is evaluated by considering the execution time and the energy consumption of the two main tasks: pre-processing and classification. The OpenMV Cam H7 Plus consumes 0.8 W in active mode. The result is summarized in Table 4.8. The pre-processing time remains the same for both architectures because this task does not use DNN models. However, the top camera pre-processing time is slightly higher than side cameras because of the image binarization operation. MobileNetV2 outperforms SqueezeNet for both top camera and side cameras, consider-

ing the classification time and the energy consumption. As a result, MobileNetV2 needs only 240 ms and 233 ms for the top camera and the side camera, respectively, to process one image and classify the object. Consequently, it consumes 192 mJ and 186 mJ for the top camera and side camera processing, respectively. Moreover, the industrial moulding machines, supported by the developed system, take about 20 seconds to produce two moles of 8 pieces. Thus, 240 ms to classify one piece is enough to guarantee the continuous and smooth operation of the production lines.

4.8 Conclusions

High-accuracy product quality inspection is a fundamental step in any manufacturing process. It permits to boost in production yield and reduces production costs. By using machine learning techniques, developers can automate and make the process real-time. This chapter presents the development and study of an innovative sensor system for automatically inspecting on-edge the quality of objects in large-scale production. The system exploits three smart cameras trained to detect and classify different anomalies in the components. Two different DNN models - namely the MobileNetV2 and the SqueezeNet - are trained and assessed, showing an accuracy of 99% and 98%, respectively. Thanks to the learning model optimization, the system can achieve respectively 5 FPS and 2 FPS for the two learning models while executing the evaluation on the edge of resource-constrained smart cameras. Future work will investigate new and innovative training techniques, like NAS, and enhance some features by integrating continuous learning functionalities to evolve the model automatically and extend the set of anomalies detectable by the system. Moreover, the system's efficiency will be improved by using cameras to detect only relevant information directly from the imager, avoiding useless processing of uninteresting pixels.

Table 4.8: Execution time and energy consumption of the pre-processing and classification tasks for the top camera and the side cameras.

160x160px	**Top Camera** binarization=True				**Side Camera** binarization=False			
	Preproc. (ms)	Class. (ms)	Total (ms)	Total Energy (mJ)	Preproc. (ms)	Class. (ms)	Total (ms)	Total Energy (mJ)
MobileNetV2	125	115	240	**192**	118	115	233	**186**
SqueezeNet	125	390	515	412	118	390	508	406

Chapter 5

Online Learning in Resource-constrained Devices

5.1 Online Learning Algorithms for Gesture Recognition and Image Classification

TinyML in IoT systems exploits MCUs as edge devices for data processing. However, traditional TinyML methods can only perform inference, limited to static environments or classes. Real-case scenarios usually work in dynamic environments, thus drifting the context where the original neural model is no longer suitable. For this reason, pre-trained models reduce accuracy and reliability during their lifetime because the data recorded slowly becomes obsolete or new patterns appear. Continual learning strategies maintain the model up to date, with runtime fine-tuning of the parameters. This work [1] compares four state-of-the-art continual learning al-

[1] The work presented in this section has been published in the following paper:

- Avi, A., Albanese, A., and Brunelli, D. (2022, July). Incremental online learning algorithms comparison for gesture and visual smart sensors. In *2022 International Joint Conference on Neural Networks (IJCNN)* (pp. 1-8). IEEE.

gorithms in two real applications: i) gesture recognition based on accelerometer data and ii) image classification. Our results confirm the systems' reliability and the feasibility of deploying them in tiny-memory MCUs, with a drop in the accuracy of a few percentage points concerning the original models for unconstrained computing platforms.

5.1.1 Introduction

IoT technology relies on the massive use of cloud computing resources to elaborate data generated by distributed objects and sensors. Improvements and more efficient applications have already been demonstrated by shifting the attention from cloud to edge and distributing the computation along the IoT chain, including gateways and nodes [111, 4, 10]. Using MCUs for intensive data elaboration can re-modulate part of the power consumption, which is crucial for these tiny devices. MCUs can optimize the data transmission and the data elaboration, which leads to the generation of more compact and meaningful information, and decreases the traffic of data on IoT networks.

Among the different opportunities opened by edge computing, machine learning on tiny embedded systems is gaining momentum. TinyML explores various types of models that can run on small, low-powered devices like microcontrollers for applications that require low latency, low power, and low bandwidth model inference. However, tiny devices feature limited memory and computing capability, which makes challenging the usage of ML models in edge devices. Thus, developing ML models of a few hundred Kbytes capable of keeping high accuracy while running MCU-enabled devices is still challenging. Another challenge is maintaining and upgrading deployed applications, which can be complex if devices are located in impractical positions. Maintenance can be required for damage, upgrade, malfunction, and, more and more frequently, for neural network (NN) model updates permitting the IoT device to evolve and correct its output.

Recently, continual learning (CL) systems have been introduced to update NN models on the MCU in real-time while the inference application runs. Current CL methods exploit continuous control over the error committed by the prediction to guarantee stable accuracy

and reliable output. Such methods present some drawbacks, and the most important is the catastrophic forgetting [186]. This problem consists of a remarkable reduction of the classification accuracy of already learned classes while the system learns an additional class from new data. Online or continuous learning applied to MCUs is a recent topic receiving growing attention, mainly because of the need to generate intelligent IoT systems that automatically self-upgrade, reducing the required maintenance.
In this work, we test and compare the performance of state-of-the-art continual learning algorithms and propose changes to fit MCU constraints. We did several tests using two different case studies, both targeting the classification of various types of data. The first is a typical gesture recognition application that classifies accelerometer data streams. We used an SMT32 Nucleo F401-RE equipped with an accelerometer shield as a test platform. This case study can be extended to other real-life applications such as industrial condition monitoring or anomaly detection [187, 188]. The second case study compares various CL algorithms to classify instances from the MNIST dataset [189]. The tests are done on an OpenMV Cam H7 Plus, which is on an ARM Cortex M7. Even this experiment can be easily extended in real-life applications such as visual inspections of products in a manufacturing process. The performance comparison of CL algorithms is made starting on pre-trained models. All the algorithms use a small framework developed on the MCUs to modify and change the parameters of some layers (usually the last one) of the pre-trained model runtime. This makes the MCU an inference machine with training capabilities only on the selected layers. This work presents the following contributions:

- The framework implementation for continual learning algorithms on STM32 MCUs and OpenMV Cam H7 Plus with micro-python interface. The framework extends the X-CUBE-AI expansion pack developed from STM for performing inference. CL algorithms use the error committed by the prediction to update the weights, save them in an array, and use the update rule defined by the selected CL algorithm.

- Improvements of the most recent algorithms in the literature. The methods have been modified to apply backpropagation on the weights that depend on batches of incoming data;

- Comparison of state-of-the-art continuous learning algorithm

performance in two case studies.

5.1.2 Related Works

TinyML has registered an impressive rate of scientific literature in the last few years. So far, the main topics are the implementation of frameworks for optimal compression and the deployment of models for efficient inference. For example, MCUNet [183] was one of the first to achieve high accuracy on off-the-shelf commercial microcontrollers, while other methods present deep compression with pruning [190] or SRAM-optimized approaches useful for the entire workflow in an ML model, including classification, porting, stitching and deployment [191].
In recent times, TinyML has expanded, and a relevant new branch related to Tiny online learning (TinyOL) has gained increasing attention. CL has already been explored for classic server and parallel architectures, but only in the last period focused on resource-constrained platforms. As of now, there already exist some well-performing strategies and frameworks like TinyTL [192], Progress & Compress [193], TinyOL [194], and Train++ [195].
Furthermore, CL has already been successfully applied in a domain of interest for our study: image classification. For instance, [196] explores the usage of developmental memories for the damping of forgetting, and [197] applies CL by using transfer learning and k-nearest neighbour. While applications of CL for pattern recognition on accelerometer sensors are still pretty new, this field has been explored only in some standard TinyML applications [198]. The studies mentioned above focus on creating memory and energy-efficient algorithms, and on backpropagation management and parameter manipulation. The work presented in [199] applies CL on the edge with an exploration of federated learning, a method used for training distributed devices where ML models are trained locally and then merged into a global model. However, CL systems are not only related to training algorithms. For instance, Imbal-OL [200] is a pre-processing technique that aims to remove unbalances of real-time data streams that are often present in real-life scenarios. The plugin can be added in between the input stream and the OL system to perform elaboration on the inputs and quickly adapt to changes while also preventing catastrophic forgetting.
CL strategies able to contrast catastrophic forgetting can be grouped

into 4 categories [201]. The first consists of architectural approaches that base their ability to overcome forgetting on the dynamics modification applied to the model's architecture. An example is the creation of dual memory models like the CWR algorithm, where one weight matrix is used as short-term memory and another as long-term memory. The second category contains regularization approaches. These methods rely on adding some loss terms in the error computed from inference. The goal is to use these loss terms to redirect the backpropagation to maintain previously gathered knowledge while adapting to changes. Some of the most used and best-performing strategies of this kind are Elastic Weight Consolidation (EWC) [186], Learn Without Forgetting (LWF) [202] and Synaptic Intelligence (SI) [203]. The last two strategy groups are the rehearsal and generative replay approaches. This study does not consider them because they require a high amount of memory and computations, thus unsuitable for TinyML applications. Rehearsal approaches are based on periodic refreshes of old data, which help the model avoid drifting from the original context. On the other hand, generative replay methods require using generative models to artificially generate past data to ensure that past knowledge is not forgotten.
CL systems can also be divided into two scenarios: multi-task (MT) and single-incremental-task (SIT). The first consists of learning tasks that are produced one by one in a controlled way without forgetting the previous ones. SIT is still an unexplored scenario and consists of neural networks that continuously expand their learning space that depends on the input data. SIT scenarios can lead to flexible systems that can add new classes to the NN design. However, the obtained systems are usually affected by catastrophic forgetting, which is a phenomenon that occurs when the knowledge related to past tasks is replaced with the newly-learned knowledge. Our study proposes the implementation of some regularization approaches aforementioned in SIT scenarios that can overcome catastrophic forgetting and are directly connected to the last layer of the NN model of interest [204].

5.1.3 System Design

We started from the release of the TinyOL implementation presented in [194]. The system can be attached to an already trained

classification model to expand its learning capabilities. It permits both to learn never-seen classes and to fine-tune the parameters in the last layer in real-time. Differently from the system developed in [194], the proposed solution completely substitutes the final layer of the model in which the classification is performed, and it is attached to a truncated version of the pre-trained model, called the *frozen model*. During the continual learning procedure, if the OL system detects a new class, it expands the dimension of the last layer, thus creating new weights and biases dedicated to the classification of the new class. In this study, because of the limited resources of the MCU, only the last layer is modified. The training is performed every time a new sample is received, with the aims of: i) fine-tuning the model to recognize and adapt to changes in old classes (known as New Instances settings, NI), ii) learning new patterns belonging to new classes never exposed to the system (known as New Classes setting, NC), iii) minimize the catastrophic forgetting [205]. We propose two examples characterized by New Instance and Classes (NIC) type data streams, meaning that the data contains new categories and new instances of old classes. As described in [204], this setting better represents a real-world application.
The OL system proposed in this work consists of the frozen model and the OL layer. The main purpose of the frozen model is to perform the feature extraction of the input and later feed the elaborated data to the classification layer, which can change shape according to the classes it encounters. The second component is the OL layer, composed of a matrix that contains the weights and an array for the biases. The width and height of the weight matrix are directly dependent on the number of classes that the model can predict and the length of the last layer of the frozen model, while the shape of the bias array depends only on the number of known classes. Note that the initial values of the OL layer are not zeros or random values but just a copy of the original classification layer computed during the frozen model training.

Continuous Learning Algorithms

The general structure of the proposed systems is similar across all the algorithms implemented. The training is performed in real-time in supervised mode, meaning that each time a data array and a ground truth label are received, a training step is performed on the

OL layer. After this step, the data is discarded, saving memory in the device.
As shown in Figure 5.1, the data array and the label sent to the MCU are immediately elaborated by the frozen model. Before the OL layer starts the training, the label is checked. The goal is to control if the label received is already known by the system or not. If the label is an unknown class, the shape of the OL layer is enlarged, with a new row to the weight matrix, and a new cell to the bias array. This is possible because the OL layer is allocated at the boot in the RAM of the MCU, while the frozen model is usually stored in the flash memory of the device. After the feature extraction is performed by the frozen model, the output is fed to the OL layer. Here, the classification is computed with the formula $output = W \cdot frozen\_out + B$ and a softmax function applied to the output. The prediction is then compared with the ground truth label and, based on the error, the weights and biases are updated according to the specific algorithm rules. This work focuses on the performance evaluation of some state-of-the-art algorithms designed for unconstrained platforms. Our contribution also includes the optimization of low-memory microcontrollers on the selected MCUs. The algorithms implemented are the standard NN training methods presented in [194]. In detail, they are: i) a variation of TinyOL, namely TinyOL v2; ii) the CWR algorithm initially proposed in [206] and later improved as CWR+ in [204], and iii) the algorithm LWF shown in [202, 204].

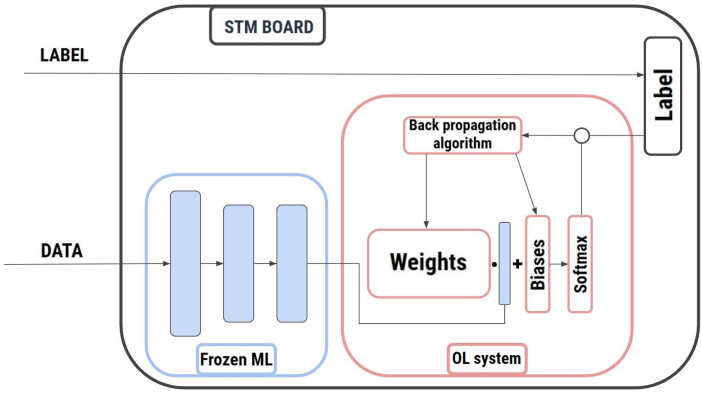


Figure 5.1: Basic block diagram of the tinyOL system.

TinyOL

TinyOL exploits the standard approach used in NN training but is applied in real-time contexts [194]. Once the data passes through the frozen model, it arrives at the OL layer. At first, the inference is performed, and later the training is done. The method uses gradient descent applied to the loss function, and a cross-entropy applied on the prediction and the ground truth label. Notice that the number of classes known from the OL system can change during runtime depending on the labels sent to the system; thus, the memory required for the OL layer can change unpredictably.
To overcome catastrophic forgetting, a variation of this method is implemented. We propose to exploit the information coming from a small batch and not just from the last sample read. In this case, the system requires the allocation of another weight matrix and bias array of the same size. The idea is to use the OL layer for inference, and each time a sample is received and elaborated by the model, its backpropagation is not applied directly to the OL layer but rather added in new containers called W and B (one for weights, the other for biases). When k samples are received and elaborated by the system, the OL layer can be updated using the average of the backpropagation data saved in W and B. Note that the user tunes the value k that defines the batch size. The updates of the weights at each step become:

$$W_{i,j} = W_{i,j} + \alpha(y_i - t_i) \cdot x_j \tag{5.1}$$

$$B_i = B_i + \alpha(y_i - t_i) \tag{5.2}$$

At the end of each batch, the backpropagation on the training weights is:

$$w_{i,j} = w_{i,j} - \frac{1}{batch\_size} \cdot W_{i,j} \tag{5.3}$$

$$b_i = b_i - \frac{1}{batch\_size} \cdot B_i \tag{5.4}$$

Where y_i is the prediction obtained from the OL layer, t_i is the true label, α is the learning rate (tuned by the user), and $w_{i,j}$ and b_i are the weights and the biases of the OL layer, respectively. The TinyOL method is a straightforward implementation of online training. It requires an amount of memory that depends only on the size of the weight and bias matrices, which depend on the number of

classes known and the height of the last layer of the frozen model. The method allows the model to change its parameters each time new data arrives without any constraints. This is why it cannot efficiently contrast the catastrophic forgetting.
TinyOL with mini-batches uses the same regularization approach but with double the memory. Unlike the TinyOL, this method tries to average the backpropagation over the last k samples received. With a small value of k, it is not contrasting the catastrophic forgetting but rather is trying to optimize the weight and bias variations to create a model that better predicts the outcome of batches. With a larger value of k, it can be possible to update weights, thus maintaining control over the forgetting.

1. The method **TinyOL v2** is a modified version of the original TinyOL. In this case, it tackles the catastrophic forgetting of old classes by not updating their weights and biases. This algorithm behaves as the TinyOL method except that the backpropagation is applied if the weight or bias is related to a new class. Then, the backpropagation becomes:

$$w_{i,j} = w_{i,j} - \alpha(y_i - t_i) \cdot x_j \tag{5.5}$$

$$b_i = b_i - \alpha(y_i - t_i) \tag{5.6}$$

$$where\ i = p, p+1.., n\ and\ j = 0, 1, .., m$$

   In this case, the iterator i starts from the value p, representing the first unknown class; thus, the update is performed only on the newest weights. The TinyOL v2 is implemented with mini-batches to overcome catastrophic forgetting.
   In conclusion, TinyOL v2 requires the same memory as the original TinyOL method but modifies only a portion of the OL layer to remove the catastrophic forgetting effect. However, this approach is ineffective in fine-tuning the model when it tries to learn new patterns of the original classes. Additionally, the method does not optimize the classification layer for the prediction but separates its behaviour into two parts where one stays updated while the other always remains the same, thus the weights of the last layer are not working together for the prediction. The version TinyOL v2 with mini-batches requires an amount of memory allocated slightly less than TinyOL mini batch because the matrix W and B are of shape $(n-p) \times m$ and $(n-p) \times 1$.

2. **LWF** is a regularization approach that overcomes catastrophic forgetting by balancing the backpropagation with new and old knowledge. This method requires the usage of two classification layers (each with a matrix of weights and an array of biases). The first one called *tl* (training layer), is continuously updated and performs the actual OL system inference. The second one called *cl* (copy layer), is the copy of the original frozen model classification layer. The idea of the algorithms is to perform the inference with both layers and then apply a loss function that depends on both outputs and the ground truth label. The gradient descent backpropagation is then implemented to this loss function:

$$\mathcal{L}_{LWF}(y_i, z_i, t_i) = (1 - \lambda) \cdot \mathcal{L}_{cross}(y_i, t_i) + \lambda \cdot \mathcal{L}_{cross}(y_i, z_i) \tag{5.7}$$

Where y_i is the prediction array obtained from the layer *tl*, z_i is the prediction array obtained from the layer *cl*, t_i is the ground truth label, and λ is the variable weight that defines which prediction has more relevance. In this algorithm, the loss function (5.7) is composed of a weighted sum of the cross-entropy applied to two predictions, where the first part is related to the layer *tl*, which is always updated, and the second part is related to *cl*, which represents the original model. The role of λ is extremely important because it defines which prediction can obtain more decisional power in the classification. The value of λ, as mentioned in [204], cannot be maintained constant, but rather change together with the evolution of the training. With this, the LWF algorithm is a continuous balance between the continual learning and the original behaviour. This application shows experimentally that evolving λ with a function proportional to the number of predictions performed is a good solution. Thus, the update of the loss function weights is the following:

$$\lambda = \frac{100}{100 + prediction\_counter} \tag{5.8}$$

For the sake of simplicity, this implementation follows the modification applied in [204], where the loss function (5.7) is computed with both components being cross-entropy, and the other knowledge distillation loss.

A second version of the algorithm is proposed. The variation updates the values of *cl* every k training performed. This allows us to have a model that is a bit more flexible concerning the LWF algorithm, where the two extremes are balanced (training layer and copy layer). The balancing is performed between the continually updated layer and a layer stopped in time (where its values are updated less frequently). In this case, the experimental lambda function found is defined by (5.9).

$$\lambda = \frac{batch\_size}{prediction\_counter} \tag{5.9}$$

Both the implemented LWF methods use the same amount of memory, which consists of two matrices of size $n \times m$ and two arrays of size $n \times 1$. This algorithm is more computationally expensive because it performs two predictions, which is one of its drawbacks. Nevertheless, the proposed method allows the model to overcome the problem of catastrophic forgetting.

3. **CWR** is an approach that exploits the usage of two classification layers together with a weighted backpropagation method. The first one, called *tl* (training layer), is updated every training step, and the second one, called *cl* (consolidated layer), is updated once every batch. During a training step, the inference is done only once with the *tl*, and its weights and biases are updated with the standard TinyOL method. The breakthrough of the algorithm takes place at the end of a batch with the following rule, which is applied to both weights and biases of the layers:

$$cw_{i,j} = \frac{cw_{i,j} \cdot updates_i + tw_{i,j}}{updates_i + 1} \tag{5.10}$$

$$tw_{i,j} = cw_{i,j} \tag{5.11}$$

Where $tw_{i,j}$ are the weights and biases of the training layer, $cw_{i,j}$ are the weights and biases of the consolidated layer, and $updates_i$ is an array that behaves as a counter of labels encountered.
The idea is to have a layer that changes slowly during the training and another is continuously updated. The layer *tl* behaves as a short-term memory because it gets reset every

time a batch ends, starting the training from a new point. On the other hand, the layer *cl* behaves as the long-term memory because it never gets cleaned, and its weights and biases are updated using the *tl*. The update of the *cl* layer is the particular backpropagation rule that allows the system to fuse slow memory and updated memory together. The method uses a weighted average between weights where the weighting factor depends on the number of times a specific label appeared in the training batch. Note that the method requires the inference computation only from the *tl* layer, while the other is never used. This is because while training the algorithm has no benefit in performing inferences with the *cl* layer in training mode. An inference with this layer is done only if a prediction is requested, which happens only during testing.
CWR requires twice the memory of the TinyOL method because it allocates two matrices and two arrays of size $n \times m$ and $n \times 1$. The number of computations during training can double if a prediction is requested. Most times, the method executes only one inference per sample. When requested, the method also starts the *cl* prediction, which is the most accurate of the two.

Case Studies

To test and evaluate the algorithms described above, we used two applications. The first (experiment A) uses a Nucleo STM32 - F401RE paired with a Nucleo shield IKS01A2 (Figure 5.2), equipped with a 3D accelerometer sensor. This example aims to use an NN model on the data coming from the accelerometer to classify letters written in the air with the MCU mounted on the user's hand. The NN model is initially trained to recognize the vowels, later the OL system is attached to the model allowing it to learn three new letters B, R, and M. The second application (experiment B) uses an OpenMV cam (Figure 5.2) as a visual smart sensor. It is based on the STM32H7 and permits the implementation of vision-based applications at low power consumption. This application uses a CNN model that was initially trained to recognize the first 6 digits from the MNIST dataset [189]. The OL system teaches the model to recognize the remaining digits correctly.

1. **Dataset Acquisition.** In experiment A, the dataset is ac-

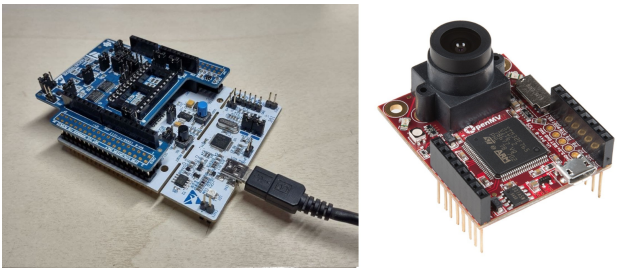


Figure 5.2: SMT32 F401RE paired with acclelerometer shield on the left - OpenMV camera on the right.

quired with the same hardware described above connected via USB to a laptop. The data is streamed in real-time via UART (USB cable) to the laptop, saving the data received in a text file. The acquisition of each letter lasts for 2 seconds, the MCU is set to work on a reading frequency of 100 Hz, making a single letter sample composed of 3 arrays (i.e., X, Y, Z accelerations) of 200 values each. All the letters in the dataset were recorded by the same user. The letters are always written in capitals following the same general path but with accentuated characteristics to make the dataset more similar to a NIC scenario (new classes and instances). The dataset is built on understanding 8 different letters, which are the original vowels A, E, I, O, and U, and the additional consonants B, R, and M. Each vowel has a total of 560 samples, while the consonants have 760 samples each. The two groups' unbalanced samples are necessary because the vowels dataset is used in two pieces of training. The first portion is used for training the frozen model, which classifies only vowels. The remaining portion is added to the OL system training dataset, which contains the letters B, R, and M. The final shapes of the two datasets are 881 samples for the training of the frozen model and 4249 samples for the continual learning application. To make the data usable by the model, each sample is reshaped from a matrix of size 3×200 in an array of shapes 1×600. Once the datasets are correctly generated, they are also shuffled separately.
The application on the OpenMV camera relies on the usage of the well-known MNIST dataset [189]. The only pre-processing

that is applied is the separation of the dataset into two groups called *low_digits* and *high_digits*. The first group represents the digits from 0 to 5, which are used for the training of the CNN frozen model, while the remaining group represents the rest of the digits used for the OL training. The dataset used for the training of the frozen model contains a total of 36017 samples, while the dataset used for the OL training contains 5000 samples (500 for each digit). The reduced amount of samples in the dataset for the OL application is due to the time taken from the method itself. Anyway, the size selected is enough to guarantee fair results.

2. **Training and evaluation.** The model used for the accelerometer application is an NN classification model with low complexity and a low number of layers. The model structure is the following: input layer with 600 nodes, two hidden layers with 128 nodes and ReLu, output layer with 5 nodes, and Softmax. The frozen model is trained locally on a laptop using Tensorflow where the used optimizer is *Adam*, and the loss function is *categorical cross entropy*. The model is trained for 20 epochs with a batch size of 16. The testing shows a final accuracy of 96.83%. Another important step is the exportation of the trained model file, which is done in two versions. The first version is the simple exportation of the original model without modifications, while the second is the exportation of the truncated version (frozen model), where the last layer is removed. This is necessary because the OL system requires total control of the weights and biases of the last layer. Removing these values from the model file and saving them in a text file in matrix form makes it possible to reload those in the code as matrices later. This allows the user to perform the standard inference from the model and later propagate the output through the last layer's weights, which are now accessible and editable.
   The second application uses a CNN, which is more suited for image classification. The model structure is the following: two Conv2D 8 filters with Relu, MaxPooling 2x2, two Conv2D 8 filters with Relu, MaxPooling 2x2, Dropout 0.25, flatten, and Dense to 6 with Softmax. The frozen model is trained with Tensorflow with *Adam* optimizer, *categorical cross entropy*

as loss function, 30 epochs, and a batch size of 64. The same exporting procedure presented above is performed. Note that, due to the limited memory on MCUs, pruning and quantization on the trained model are always suggested. In this case, the model structure does not require compression, but we decided to use it anyway to demonstrate the performance of these models.

3. **TinyOL Implementation.** The OL layer is composed of one matrix for the weights and an array for the biases. These two containers are initialized at the beginning of the code. Their initial values are copied from a text file generated from the training step. In this way, it is possible to have a classification layer that starts exactly from where the Tensorflow training stopped. The matrix and array for experiment A have an initial shape of 5×128 and 1×5, respectively. The matrix and array for experiment B have an initial shape of 6×512 and 1×6, respectively. Note that in both applications, during training, each time a new class (letter or digit) is found, the OL system adds one row to the weights and one cell to the biases. This increases the allocated RAM, and it is important to ensure that the full capacity is not met.
   The pseudo-test is done to test the performance of the CL algorithm of interest at running time. It is crucial to begin the test once a portion of the dataset has been processed to accurately represent the training method. This type of testing better resembles a real-world application, where the OL system is deployed in an environment for an indefinite period, and its performance is checked online during runtime. In experiments A and B, the pseudo-testing starts when the training surpasses the 80% of the dataset available.

4. **Evaluation Metrics.** We measured the accuracy, training step time, and the maximum allocated RAM, to evaluate the performance of the CL algorithms. The accuracy indicates how much the model can predict incoming data and adapt to new data. Considering that this work aims to prove the feasibility of deploying CL algorithms in MCUs, the accuracy can be enough to evaluate the models' goodness. The accuracy is computed with new data during testing; however, cross-validation can be considered for a more robust test. The

training step time gives the period that the continual learning algorithm needs to compute a prediction and update its weights and biases. This metric assesses the algorithm's responsiveness. Finally, the maximum allocated RAM permits a tradeoff between the usage of resources on the MCU and the achieved accuracy. These metrics allow a proper and efficient selection of the best algorithm running on embedded systems.

5.1.4 Experimental Results

Before benchmarking the proposed algorithms, we show a comparison between results achieved on a laptop and those using the MCU. It is necessary to confirm that the performance is comparable and the algorithms are correctly optimized for the microcontrollers and capable of running online training reliably. After this, the results from the application on the OpenMV camera are shown, and the algorithm's accuracy is compared when new classes are added online. The evolution of all the relevant parameters is stored from both devices and compared. Parameter histories from both devices show the same behaviour with a couple of exceptions that show a minor magnitude difference for just a few steps. From this test, it is possible to conclude that the MCU behaves in the same way as the laptop, proving that a device with such limited resources can be considered reliable and can be directly compared with the performance of more powerful devices.
Table 5.1 summarizes the performance of all algorithms from experiment A. Each test is performed with the same frozen model and dataset in the same order. The first row clearly shows that all the methods perform rather well, with the lowest accuracy being 86.13% (10.7% less than the original Tensorflow training due to the addition of the OL system). This means that a dataset of 4000 samples with approximately 500 samples for each class is enough for adequately training a model on three additional classes. Another important parameter is the inference time. The frozen model always behaves in the same way with a total inference time of 10.65 ms, no matter the strategy adopted. However, the inference time for the OL layer changes depending on the algorithm used. The slowest method is the LWF algorithm, which on average, requires more than double the time of all the other methods because it performs double the predictions for each training step, thus doubling the computations.

All the other methods are very close, and the small differences are mainly due to different behaviours at the end of a batch. Another useful comparison is the RAM allocation. Table 5.1 confirms that the two lightweight methods are TinyOL and TinyOL v2, which allocate only one matrix and one array. The remaining methods use a similar amount of memory with a small variation of 100 bytes. Note that the Nucleo STM F401-RE MCU's total available RAM is 96 kB.
Better conclusions can be drawn from the plot in Figure 5.3 which

Table 5.1: Accuracy, average training step time, and memory allocated during training for all algorithms - STM application.

	TinyOL	**TinyOL batches**	**TinyOL V2**	**TinyOL V2 batches**	**LWF**	**LWF batches**	**CWR**
Accuracy (%)	86,13	86,26	87,98	87,98	87,61	86,50	**88,47**
Training Time (ms)	**0,99**	1,54	1,03	1,11	3,45	3,26	2,11
Max RAM (kB)	**26,10**	29,80	**26,10**	29,80	29,90	29,90	29,90

displays the prediction accuracy for each method. The bar plot shows that all methods successfully perform OL training and maintain high accuracy. The method that shows the best overall accuracy is the CWR algorithm; however, the difference is small. Thus, it is taken into account the inference time and the memory used. In general, all classes are correctly integrated into the OL model, and only some classification mistakes occurred in the letters R and B because of their similar shape. Another important trend present in all algorithms is how the accuracy is spread quite uniformly among all classes. This confirms that all methods can perform continuous training and include new classes in their model structure.
The overall performance of the OL systems compared to the frozen model is reduced. This suggests that the accuracy of the original model is sacrificed to balance the system. This behaviour can be tested even more by enlarging the dataset, which would help to understand if the model training has been early stopped.
Another important aspect concerns the *catastrophic forgetting*. No method shows a strong effect against or in favour of it. Even the TinyOL method, which is considered the most vulnerable to the phe-

nomenon, does not show poor behaviour, meaning that the catastrophic forgetting in this specific application is not particularly severe. The phenomenon is probably dumped by the randomization of the dataset, which allows the models to refresh old data and learn new classes continuously.

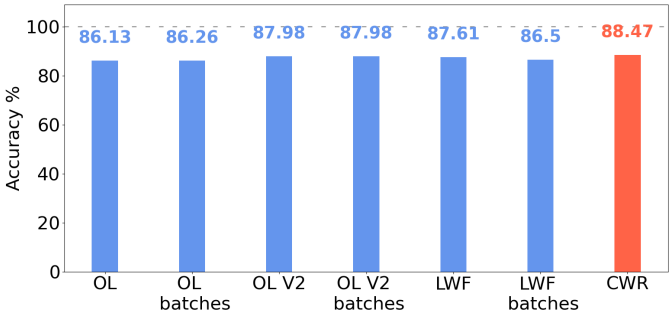


Figure 5.3: Accuracy of each strategy used - STM application.

Concerning experiment B, the most relevant results are summarized in Table 5.2. In this case, the test shows a higher accuracy compared to the previous experiment, most probably due to the high-quality dataset used. The inference performed by the frozen model stays constant for all methods and is 15.88 ms. The increase of inference time for the OL layer in this application is given by the increased complexity of the frozen model. Instead, the inference time for the CL strategies is a bit different. All the methods that do not use batch updates are faster than their respective batch methods. Further conclusions can be drawn from the bar plots in Figure 5.4. In this case, the lowest accuracy is higher than the previous application, and in particular, the OL models have a much less drop in accuracy when compared to the Tensorflow training, from 99.35% to 93.09%. Also, in this experiment, the catastrophic forgetting is not severe. The most sensible method for the phenomenon (the OL method) does not show any problem due to the continuous refreshment of the old data.
Another important comparison can be made between the methods and their respective implementation with batches. The accuracy difference in Table 5.2 is minor, but better conclusions can be drawn with Figure 5.5, where the overall method accuracy is tested at the

Table 5.2: Accuracy and average training step time for all algorithms - OpenMV application.

	TinyOL	TinyOL batches	TinyOL V2	TinyOL V2 batches	LWF	LWF batches	CWR
Accuracy (%)	94,39	95,40	94,39	93,09	95,20	94,99	**95,70**
Training step time (ms)	3,02	3,35	**2,13**	4,24	4,86	5,20	3,32

variation of the batch size. It is clear how the increase in batch size does not improve the model's accuracy. This suggests that the learning improvement (particularly for TinyOL and TinyOL v2) from just one sample is more significant than the averaged info obtained from a bigger group.

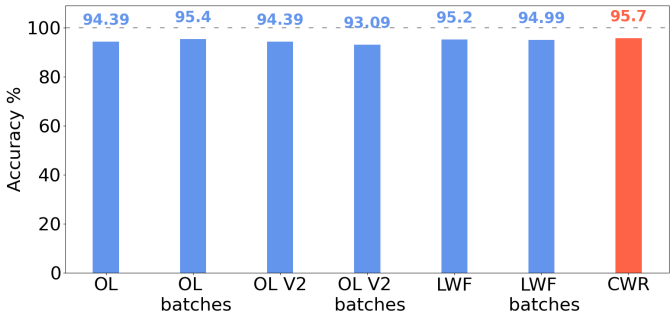


Figure 5.4: Accuracy of each strategy used - OpenMV application.

5.1.5 Conclusions

This work explores online learning on microcontrollers in two different scenarios concerning the analysis of accelerometer data and the classification of images. We adopted four state-of-the-art continual learning strategies with some improvements from our side. We showed that continual learning on small MCUs is a feasible solution for contrasting dynamic contexts, and the results obtained can generate self-sustainable and adaptable models. Our results demonstrate the capability of continual learning strategies to adapt to the extension of 4 new classes in the case of image classification or 3

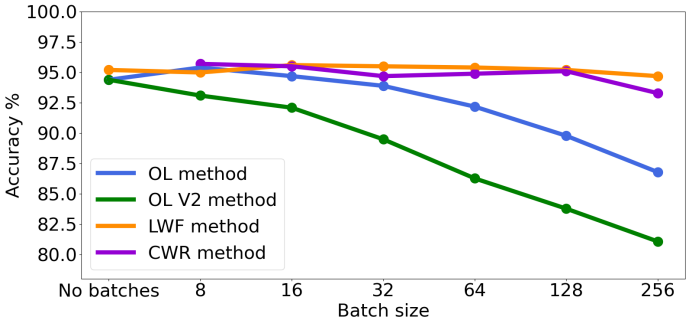


Figure 5.5: Accuracy of each class at variation of batch size - OpenMV application.

classes in the case of gesture recognition. Models trained in these conditions can maintain high performance with an acceptable drop in accuracy of 10.7% for the accelerometer example and 6.3% for the MNIST digits recognition.
The obtained results demonstrate the potential of this type of technology, especially if applied to small and constrained devices. Continual learning applied in tinyML is a good alternative and an improvement to the standard approach of train-and-deploy because it permits flexible, self-adapting, and self-updating systems.

5.2 Unsupervised Online Learning on Edge Devices

Traditional tinyML systems are widely employed because of their limited energy consumption, fast execution, and easy deployment. However, such systems have limited access to labelled data and need periodic maintenance due to the evolution of data distribution (i.e., context drift). Continual machine learning algorithms can enable CL on embedded systems by updating their parameters, addressing context drift, and allowing neural networks to learn new categories over time. However, the availability of labelled data is scarce, limiting such algorithms in supervised settings. This contribution [2] over-

[2]The work presented in this section has been published in the following paper:

- Poletti, G., Albanese, A., Nardello, M., and Brunelli, D. (2023, September). Tiny Neural Deep Clustering: An Unsupervised Approach for Con-

comes this limitation with an alternative approach which combines supervised deep learning with unsupervised clustering to enable unsupervised continual machine learning on the edge. The system is deployed in an OpenMV Cam H7 Plus and tested with the MNIST dataset reaching a classification accuracy of 92.3% and a frame rate of 44 FPS.

5.2.1 Introduction

TinyML systems are trained on a fixed dataset and assume that the data distribution does not change over time. In many real-world applications, the data distribution can evolve, leading to a problem known as context drift due to changes in the environment, user behaviour, or underlying data distribution. Context drift can be a challenging problem for ML models deployed in real-world applications, where the input data may change over time. If a model cannot adapt to the changing context, its performance can suffer, resulting in incorrect predictions or decisions. A solution to overcome context drift is CL. A different approach consists of applying CL to resource-constrained devices (e.g., MCUs) creating intelligent self-updated IoT systems with high flexibility requiring low maintenance in challenging scenarios [207, 208, 209]. Common state-of-the-art CML strategies, such as CWR [204], LWF [203], and tinyOL [194], can mitigate the effects of context drift; however, they need a ground truth to carry out the backpropagation for weights and biases updates. Those approaches are usually referred to as supervised learning. Thus, the incoming data must be labelled, making such systems unfeasible for real-world applications because of the unavailability of ground truths. Other works, such as Online Deep Clustering [210] and Unsupervised Continual Learning for Gradually Varying Domains [211] avoid the need for labelled data. In the former case, the authors use a combination of clustering and deep learning algorithms to implement a semi-unsupervised continual learning system. In the latter, authors use clustering as a classification system to adapt to gradually varying domains. Even though these works show successful solutions for

tinual Machine Learning on the Edge. In *International Conference on Applications in Electronics Pervading Industry, Environment and Society* (pp. 117-123). Cham: Springer Nature Switzerland.

unsupervised CL, they do not include embedded implementation, which is challenging for real-world applications. So far, only the work presented in [212] presents a solution for on-device training with a low-memory budget. However, this solution still works in a supervised setting, and during the on-device retraining, the data need the ground truth. Thus, an embedded solution for implementing unsupervised CL to get closer to real-world exploitation is still missing making this work a useful contribution to prove the effectiveness of unsupervised CL on the edge. Unsupervised CL on the edge can support many domains such as industrial visual inspection [166], surveillance [22, 213], smart agriculture [111, 10], and medical applications (e.g., histopathology) to enhance the period of use without maintenance[214][215]. In this section, we present Tiny Neural Deep Clustering (TinyNDC), an innovative approach that combines supervised learning with unsupervised clustering to enable unsupervised CL on resource-constrained devices. The MNIST dataset is used to evaluate the proposed algorithm. The first 6 digits are used to initialize the model, while the remaining 5 are used to test the online learning algorithm. Results show that TinyNDC can achieve 44 FPS and a learning accuracy of 92.3% on average among the 10 MNIST digits.

5.2.2 System Design

Combining clustering and deep learning systems can be proven a successful solution. It mixes the flexibility of clustering in adapting to different domains with the stability of DL in avoiding drift from the current domain. We propose an algorithm, namely TinyNDC that is tailored for running in an ARM Cortex M7 processor.

Figure 5.6 presents the architecture of TinyNDC and is composed of three main components:

- **Frozen model**. The static part of the model which is not updated during runtime. It can be any NN truncated in the last layer (e.g., the Softmax classifier).

- **Active model**. The dynamic part of the model which is updated during runtime. It is composed of two sub-modules, namely the *consolidated layer* and the *training layer*. The first acts as a memory to mitigate catastrophic forgetting, while the latter is used for domain adaptation. The active model

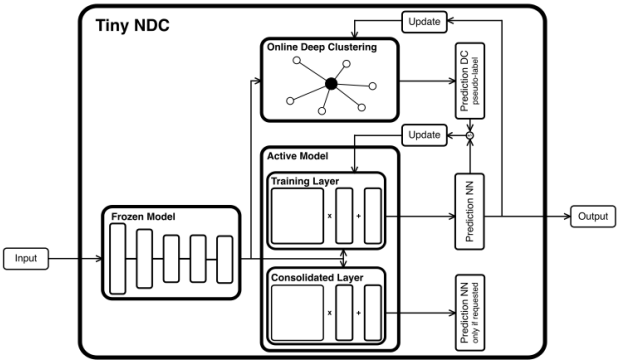


Figure 5.6: TinyNDC architecture. Data are fed to the *Frozen Model* that returns a set of features. Online deep clustering clusters the features, calculating the closest centroid, while the *Active Model* carries out the classification. Then the active model prediction is used to update the centroids of online deep clustering, and the pseudo label produced by the clustering algorithm is used to update the weights and biases of the active model.

can update its weights and biases and add new classes if the current data is not associated with an already known class.

- **Online deep clustering**. It is fed with the output of the frozen model and produces the pseudo-label estimation. This label is fundamental for updating the active model. Furthermore, the clustering can update its centroids by using the prediction of the active model. The clustering's centroids need to be initialized with a small amount of data (e.g., a hundred samples).

5.2.3 Experimental Setup

To evaluate the performance of the proposed TinyNDC algorithm, we use the MNIST dataset [189] and an OpenMV Cam H7 Plus (OMV) [3] device. The frozen model is trained to classify digits from 0 to 5, while the remaining digits were used to test the CML capabilities in an unsupervised scenario. Specifically, the dataset con-

[3] https://openmv.io/products/openmv-cam-h7-plus

sists of a total of 60000 samples. 35000 samples are used for training the frozen model, 100 for cluster initialization (10 per class), 7000 for the online learning phase, and 1000 for assessing the CML performances. Three different online learning algorithms, namely CWR [204], LWF [203], and tinyOL [194], are tested as update mechanisms for the active model. Accuracy is selected as the performance metric. CWR, LWF, and tinyOL reach an accuracy of 92.3%, 91.3% and 91.7%, respectively. The experimental setup consists of correctly identifying data shown to the OMV through a PC screen. As CWR is proved to be the best-performing algorithm it is chosen as the target update rule for performing further tests.

Frozen Model Training

TinyNDC is pre-trained in a supervised fashion on a PC to recognize digits from 0 to 5. The frozen model generation and training are carried out with the TensorFlow library and Python on the PC. We used a CNN specifically designed for image processing composed of four 2D convolutional layers. For the training phase, *Adam* is used as an optimizer, categorical cross entropy as a loss function, with a total of 40 epochs, and a batch size of 32. After the training phase, the frozen model shows an accuracy of 99.64% on average among the first 6 digits.

Centroids Inizialization

After the initialization of the frozen model, we moved to the centroid initialization, needed as a preliminary step. We used 10 samples per category, thus we extracted 100 samples from the 60000 training images. The OMV snaps a photo of each digit that shows up on the screen, feeds it to the trained frozen model for feature extraction, and sends the resulting array to the PC. Finally, the PC collects the 100 arrays, where each image feature is represented by a vector of 512 samples. The brightness of the screen is fixed. The clusters are defined by the sample labels and each centroid is calculated through the mean of all the respective cluster samples.

5.2.4 Experimental Results

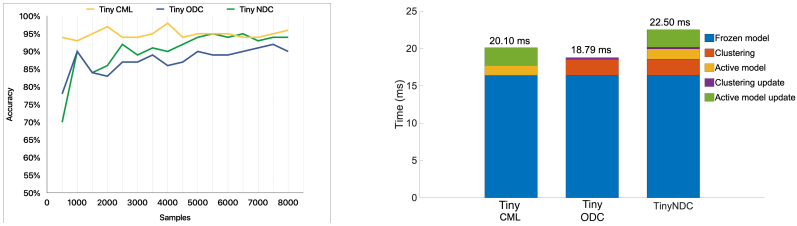
The proposed system is compared with the same system composed of the frozen and active model, *TinyCML*, in a supervised setting (i.e.,

without the clustering algorithm for estimating the pseudo-label) to monitor performance degradation in moving from supervised to unsupervised CL. Furthermore, a comparison is also conducted with the clustering algorithm *TinyODC*, which acts as an unsupervised CML system to ensure the stability and feasibility of the proposed system. In this case, the clustering algorithm can classify data and update its centroids during runtime. The comparison, presented in Table 5.3, shows that even though our system works in a more challenging setting (i.e., unsupervised), it reaches good performance almost comparable with the supervised system. Furthermore, the clustering algorithm alone performs worse than the *TinyNDC*, confirming the effectiveness of the combination of clustering and DL. Figure 5.7(a) presents the learning curve comparison considering the accuracy. It shows that *TinyNDC* starts learning with a low accuracy level compared to *TinyCML*, but, after 8000 samples, it almost reaches the learning level of *TinyCML*. This further confirms the stability of clustering and DL combination, which can reach the performance of the associated supervised learning system. It can be argued that the best performing algorithm, as shown in Table 5.3, is *TinyCML*. However, it uses a supervised learning approach that is not feasible in real-world scenarios. On the contrary, *TinyNDC* trades a mere 2% accuracy loss with the ability to work in an unsupervised fashion, making it feasible to be implemented in real-world applications.

Table 5.3: Scores comparison between supervised CML (*TinyCML*), unsupervised CL with only clustering (*TinyODC*), and the proposed solution *TinyNDC*. *TinyCML* uses an initial learning rate of 0.8, a decay rate of 2, a learning rate step size of 500, and a batch size of 16. Tests are conducted at 160 lux using 7000 samples for the training and 1000 for the validation.

Algorithm	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1-score (%)**
TinyCML	94.3 ± 1.2	95.0 ± 1.2	94.0 ± 1.2	94.0 ± 1.2
TinyODC	90.7 ± 0.6	91.0 ± 0.6	91.0 ± 0.6	91.0 ± 0.6
TinyNDC	92.3 ± 1.2	93.0 ± 1.2	92.0 ± 1.2	92.0 ± 1.2

Finally, the proposed system is characterized by measuring the execution time of each task to ensure real-time capability and possible employment in real-world applications. The *TinyNDC* tasks' execution times are compared with supervised CML (*TinyCML*) and



(a) (b)

Figure 5.7: Comparison of *TinyCML*, *TinyODC*, and *TinyNDC* for (a) accuracy learning curves, and (b) tasks' execution time.

Table 5.4: Execution times of different sections of supervised CML (*TinyCML*), unsupervised CML with clustering (*TinyODC*), and *TinyNDC*.

Section	**TinyCML (ms)**	**TinyODC (ms)**	**TinyNDC (ms)**
Frozen Model	16.39	16.39	16.39
Clustering	-	2.18	2.18
Active Model	1.38	-	1.38
Clustering Update	-	0.22	0.22
Active Model Update	2.34	-	2.34
Total	**20.10**	**18.79**	**22.50**

unsupervised CML with clustering (*TinyODC*). As shown in Table 5.4, *TinyNDC* needs more time than the other strategies to process one sample and perform the model update. On the other hand, as shown in Figure 5.7(b), most of the processing time is needed by the frozen model, which requires 16.39 ms to produce the inference result, while the active model and the clustering need only 6.12 ms for inference and update. Thus, even though *TinyNDC* uses a combination of clustering and DL, the processing complexity is not increased considerably, confirming the feasibility of using *TinyNDC* for real-time applications.

5.2.5 Conclusions

Continual learning is an active research area that aims to provide self-updating systems. This permits the usage of such algorithms in challenging scenarios where it is difficult to perform periodic maintenance. However, such systems are computationally intensive, thus too power-demanding for machine learning on the edge exploitation. This work presents TinyNDC, an initial approach to unsupervised learning on the edge by combining clustering with deep learning. The algorithm is tested on an MCU-based camera, namely OpenMV Cam H7 Plus, and evaluated against the MNIST dataset. Results show that the system achieves a frame rate of 44 FPS and a learning accuracy of 92.3%. Future work will employ the usage of a more deep update by considering the inner layers and benchmarking with real case studies.

Chapter 6

Conclusions

The IoT revolution denotes integrating sophisticated digital technologies into common devices to support different applications such as agriculture, manufacturing, healthcare, and consumer electronics. It involves the utilization of automation, artificial intelligence, big data, and wireless communication to establish smart devices and revolutionize the daily lives of individuals. IoT holds the potential for heightened efficiency, enhanced productivity, and greater flexibility. It includes robotics, intelligent machines, wearable devices, and distributed monitoring systems allowing them to support daily and repetitive tasks, permitting humans to focus on more complex and value-added problems. This improves overall productivity, reduces the likelihood of errors, and increases service quality.
Recent advancements have demonstrated that the tinyML approach is a successful solution by bringing the neural processing (i.e., the inference step) on the device, achieving an energy-efficient execution with a real-time frame rate. However, such systems present challenges such as large model deployment in constrained devices (e.g., MCUs), efficient model optimisation, and on-device learning implementation. Furthermore, tinyML systems are affected by continuous environmental changes (e.g., light, temperature, reflection), leading to unstable and obsolete systems. For this reason, such systems need periodic maintenance, such as model retraining, to ensure they follow the environmental behaviour. On the other hand, maintenance means stopping the system operation for a limited period; furthermore, such systems can be deployed in hard-to-reach posi-

tions, making their preservation an expensive and difficult task.
To drastically limit maintenance, on-device learning is a successful solution to obtain self-updated systems during run-time and avoid model retraining. It represents a paradigm shift in machine learning, allowing models to learn continuously, mimicking the human capacity for lifelong learning. Traditional offline ML applications train models to perform backpropagation on the available dataset. This allows the model to consolidate its knowledge and optimize its decision-making based on the samples inside the training batches. On-device learning applications operate on sequentially arriving data streams, similar to the real-time nature of many applications. Due to this dynamic nature, data poses problems like concept drift, in which the distribution of the underlying data changes over time. This leads to the need for a balance between learning new things and remembering what has already been learned. On-device learning permits models to learn from new data while preserving and consolidating existing knowledge. However, such algorithms are typically computationally intensive, and on-device learning on low-power and low-cost devices becomes a challenging task that must be thoroughly explored.
The tinyML integration into real-world applications is still scarce due to the development complexity and the lack of standardised tools. Furthermore, there is not a sufficient number of successful case studies that encourage companies to invest in tinyML systems. This thesis overcomes the challenges introduced by tinyML and tinyOL by providing a framework and methodologies to improve its development and integration opening its usage in various sectors. Furthermore, it demonstrates the feasibility and the effectiveness of implementing tinyML in different real use cases highlighting its benefits related to energy efficiency, real-time capability, and low-impact memory footprint. This will enhance its employment in mechatronics systems including fix and mobile robotics, IoT, and industrial applications.
In particular, **Chapter 2** presents a smart trap for pest detection suitable for apple orchards in precision agriculture. It shows the benefits of implementing tinyML in IoT end nodes by ensuring high quality of service and the platform energy neutrality with a solar panel of a few hundred centimetres. This system permits low maintenance, low cost, and optimised usage of chemicals.
Chapter 3 moves the attention to the mobile robotics field, focus-

ing on autonomous UAVs. Also, these systems with a low payload and energy resources can benefit from tinyML implementing in-situ data processing for autonomous navigation using visual cues. It is demonstrated that using object detection with deep learning algorithms on the edge does not affect the flight time thanks to the optimisation involved. Furthermore, it explores navigation under rainy conditions, providing a possible solution to counteract adversity and increase drone operability.

Chapter 4 focuses on industrial applications, in particular, visual inspection in production lines. It presents a cyber physical system composed of three MCU-based cameras running deep neural networks on the edge. It exhibits impressive performance in defect detection as well as in responsiveness, guaranteeing the smooth operation of the production line.

Finally, **Chapter 5** addresses the challenge of on-device learning on constrained devices, namely tinyOL. It provides an efficient solution to implement learning capabilities on the edge by limiting online learning only to the last few layers of a deep neural network. This trade-off permits the efficient deployment of these complex systems ensuring the successful online training of the model. Furthermore, it shows an extension by using a combination of unsupervised learning algorithms to open its usage in many real-world applications.

Overall, the use cases analysed in this dissertation provide successful solutions with tinyML and tinyOL. They exhibit good performance in accuracy and frame rate, highlighting the benefits of using the tinyML approach. This contribution can open the usage of such systems in industrial and commercial applications providing accessible and scalable methodologies that encourage companies to use such techniques.

Chapter 7

PhD Activities

During this research experience, starting from November 2020 with a research fellowship, following a PhD in November 2021, I had the opportunity to carry out the following activities.

Journal Papers

- Brunelli, D., Albanese, A., d'Acunto, D., and Nardello, M. (2019). Energy neutral machine learning based iot device for pest detection in precision agriculture. *IEEE Internet of Things Magazine*, 2(4), 10-13.

- Albanese, A., Nardello, M., and Brunelli, D. (2021). Automated pest detection with DNN on the edge for precision agriculture. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 11(3), 458-467.

- Albanese, A., Nardello, M., and Brunelli, D. (2022). Low-power deep learning edge computing platform for resource constrained lightweight compact UAVs. *Sustainable Computing: Informatics and Systems*, 34, 100725.

- Albanese, A., Nardello, M., Fiacco, G., and Brunelli, D. (2022). Tiny machine learning for high accuracy product quality inspection. *IEEE Sensors Journal*, 23(2), 1575-1583.

- Albanese, A., and Brunelli, D. (2023). Industrial Visual Inspection with TinyML for High-Performance Quality Control.

IEEE Instrumentation and Measurement Magazine, 26(8), 17-22.

- Albanese, A., Wang, Y., Brunelli, D., and Boyle, D. (2024). Analysis and Estimation with DNNs of Adverse Rainy Conditions in Visual Odometry for UAV Autonomous Navigation. *Under review in the IEEE IoT Journal.*

Conference Papers

- Albanese, A., d'Acunto, D., and Brunelli, D. (2020). Pest detection for precision agriculture based on iot machine learning. In *Applications in Electronics Pervading Industry, Environment and Society: APPLEPIES 2019 7* (pp. 65-72). Springer International Publishing.

- Avi, A., Albanese, A., and Brunelli, D. (2022, July). Incremental online learning algorithms comparison for gesture and visual smart sensors. In *2022 International Joint Conference on Neural Networks (IJCNN)* (pp. 1-8). IEEE.

- Albanese, A., Taccioli, T., Apicella, T., Brunelli, D., and Ragusa, E. (2022, September). Design and deployment of an efficient landing pad detector. In *International Conference on System-Integrated Intelligence* (pp. 137-147). Cham: Springer International Publishing.

- Torrisi, A., Doglioni, M., Gemma, L., Albanese, A., Santoro, L., Nardello, M., and Brunelli, D. (2022, September). Batteryless Soil EIS Sensor Powered by Microbial Fuel Cell. In *Annual Meeting of the Italian Electronics Society* (pp. 277-282). Cham: Springer Nature Switzerland.

- Santoro, L., Albanese, A., Canova, M., Rossa, M., Fontanelli, D., and Brunelli, D. (2023, June). A Plug-and-Play TinyML-based Vision System for Drone Automatic Landing. In *2023 IEEE International Workshop on Metrology for Industry 4.0 and IoT (MetroInd4.0andIoT)* (pp. 293-298). IEEE.

- Poletti, G., Albanese, A., Nardello, M., and Brunelli, D. (2023, September). Tiny Neural Deep Clustering: An Unsupervised

Approach for Continual Machine Learning on the Edge. In *International Conference on Applications in Electronics Pervading Industry, Environment and Society* (pp. 117-123). Cham: Springer Nature Switzerland.

- Albanese, A., Gotta, D., and Brunelli, D. (2024, September). A tinyML-based IoT Device for Advanced Shipping Monitoring. In *International Conference on Applications in Electronics Pervading Industry, Environment and Society.*

Patent

The following patent application has been submitted with a positive first draft of review:

- Albanese, A., Barchi, F., Brunelli, D., Elia, N., and Gotta, D. (2023). Method and System for Controlling a Shipment.

Experience Abroad

- Young Fellow Student at the 59th Design Automation Conference (DAC), July 2022, San Francisco, California, USA.
  Poster presentation: "UAVs Autonomous Navigation with TinyML and Incremental Online Learning Algorithms"

- Visiting PhD student at the Dyson School of Design Engineering, Imperial College London, London, United Kingdom, from September 2023 to February 2024.
  Research project: "Analysis and Estimation with DNNs of Adverse Rainy Conditions in Visual Odometry for UAV Autonomous Navigation"
  Supervisor: Prof. David Boyle

Teaching Activities

During the PhD, I had the opportunity to assist in the following teaching activities with laboratory classes:

- "Computer Science" at the bachelor's degree in "Industrial Engineering"

- "Laboratory of Internet of Things" at the master's degree in "Mechatronics Engineering"

- "Embedded Systems" at the master's degree in "Mechatronics Engineering"

Acknowledgements

The PhD student Andrea Albanese acknowledges support from TIM S.p.A. through the PhD scholarship.

Bibliography

[1] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and<0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[2] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[3] P. A. Abdalla and A. Varol, "Advantages to disadvantages of cloud computing for small-sized business," in *2019 7th International Symposium on Digital Forensics and Security (ISDFS)*, 2019, pp. 1–6.

[4] A. Albanese, M. Nardello, and D. Brunelli, "Low-power deep learning edge computing platform for resource constrained lightweight compact uavs," *Sustainable Computing: Informatics and Systems*, vol. 34, p. 100725, 2022.

[5] W. Chen, B. Liu, H. Huang, S. Guo, and Z. Zheng, "When uav swarm meets edge-cloud computing: The qos perspective," *IEEE Network*, vol. 33, no. 2, pp. 36–43, March 2019.

[6] G. Datta, S. Kundu, Z. Yin, R. T. Lakkireddy, J. Mathai, A. P. Jacob, P. A. Beerel, and A. R. Jaiswal, "A processing-in-pixel-in-memory paradigm for resource-constrained tinyml applications," *Scientific Reports*, vol. 12, no. 1, p. 14396, 2022.

[7] A. Elhanashi, P. Dini, S. Saponara, and Q. Zheng, "Advancements in tinyml: Applications, limitations, and impact on iot devices," *Electronics*, vol. 13, no. 17, p. 3562, 2024.

[8] R. Kallimani, K. Pai, P. Raghuwanshi, S. Iyer, and O. L. López, "Tinyml: Tools, applications, challenges, and future research directions," *Multimedia Tools and Applications*, vol. 83, no. 10, pp. 29 015–29 045, 2024.

[9] D. Sartori and D. Brunelli, "A smart sensor for precision agriculture powered by microbial fuel cells," in *2016 IEEE sensors applications symposium (SAS)*. IEEE, 2016, pp. 1–6.

[10] D. Brunelli, A. Albanese, D. d'Acunto, and M. Nardello, "Energy neutral machine learning based iot device for pest detection in precision agriculture," *IEEE Internet of Things Magazine*, vol. 2, no. 4, pp. 10–13, 2019.

[11] T. Polonelli, D. Brunelli, A. Marzocchi, and L. Benini, "Slotted aloha on lorawan-design, analysis, and deployment," *Sensors*, vol. 19, no. 4, p. 838, 2019.

[12] S. Mileiko, F. Ritom, R. Shafik, A. Yakovlev, and D. Balsamo, "Run-time energy and time management for intermittent lorawan communications," in *IEEE Sensors Applications Symposium (SAS 2024)*. Newcastle University, 2024.

[13] B. A. Mudassar, P. Saha, Y. Long, M. F. Amir, E. Gebhardt, T. Na, J. H. Ko, M. Wolf, and S. Mukhopadhyay, "Camel: An adaptive camera with embedded machine learning-based sensor parameter control," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 3, pp. 498–508, 2019.

[14] M. Shoaran, B. A. Haghi, M. Taghavi, M. Farivar, and A. Emami-Neyestanak, "Energy-efficient classification for resource-constrained biomedical applications," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 4, pp. 693–707, 2018.

[15] S. Motaman, S. Ghosh, and J. Park, "A perspective on test methodologies for supervised machine learning accelerators," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 3, pp. 562–569, 2019.

[16] C. Gao, A. Rios-Navarro, X. Chen, S.-C. Liu, and T. Delbruck, "Edgedrnn: Recurrent neural network accelerator for edge in-

ference," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 10, no. 4, pp. 419–432, 2020.

[17] S. Dey, K.-W. Huang, P. A. Beerel, and K. M. Chugg, "Predefined sparse neural networks with hardware acceleration," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 332–345, 2019.

[18] T. Guo, J. Dong, H. Li, and Y. Gao, "Simple convolutional neural network on image classification," in *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)*. IEEE, 2017, pp. 721–724.

[19] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[20] H. Qassim, A. Verma, and D. Feinzimer, "Compressed residual-vgg16 cnn model for big data places image recognition," in *2018 IEEE 8th annual computing and communication workshop and conference (CCWC)*. IEEE, 2018, pp. 169–175.

[21] P. Tosato, D. Facinelli, M. Prada, L. Gemma, M. Rossi, and D. Brunelli, "An autonomous swarm of drones for industrial gas sensing applications," in *2019 IEEE 20th International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*. IEEE, 2019, pp. 1–6.

[22] M. Nardello, H. Desai, D. Brunelli, and B. Lucia, "Camaroptera: A batteryless long-range remote visual sensing system," in *Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems*, 2019, pp. 8–14.

[23] T. Polonelli, D. Brunelli, and L. Benini, "Slotted aloha overlay on lorawan-a distributed synchronization approach," in *2018 IEEE 16th international conference on embedded and ubiquitous computing (EUC)*. IEEE, 2018, pp. 129–132.

[24] R. T. Carde and A. K. Minks, "Control of moth pests by mating disruption: successes and constraints," *Annual review of entomology*, vol. 40, no. 1, pp. 559–585, 1995.

[25] P. Witzgall, P. Kirsch, and A. Cork, "Sex pheromones and their impact on pest management," *Journal of chemical ecology*, vol. 36, pp. 80–100, 2010.

[26] A. Segalla, G. Fiacco, L. Tramarin, M. Nardello, and D. Brunelli, "Neural networks for pest detection in precision agriculture," in *2020 IEEE International Workshop on Metrology for Agriculture and Forestry (MetroAgriFor)*. IEEE, 2020, pp. 7–12.

[27] M. C. F. Lima, M. E. D. de Almeida Leandro, C. Valero, L. C. P. Coronel, and C. O. G. Bazzo, "Automatic detection and monitoring of insect pests—a review," *Agriculture*, vol. 10, no. 5, p. 161, 2020.

[28] S.-H. Kang, S.-H. Song, and S.-H. Lee, "Identification of butterfly species with a single neural network system," *Journal of Asia-Pacific Entomology*, vol. 15, no. 3, pp. 431–435, 2012.

[29] S.-H. Kang, J.-H. Cho, and S.-H. Lee, "Identification of butterfly based on their shapes when viewed from different angles using an artificial neural network," *Journal of Asia-Pacific Entomology*, vol. 17, no. 2, pp. 143–149, 2014.

[30] C. Wen, D. E. Guyer, and W. Li, "Local feature-based identification and classification for orchard insects," *Biosystems engineering*, vol. 104, no. 3, pp. 299–307, 2009.

[31] L. Liu, R. Wang, C. Xie, P. Yang, F. Wang, S. Sudirman, and W. Liu, "Pestnet: An end-to-end deep learning approach for large-scale multi-class pest detection and classification," *Ieee Access*, vol. 7, pp. 45 301–45 312, 2019.

[32] S.-J. Hong, S.-Y. Kim, E. Kim, C.-H. Lee, J.-S. Lee, D.-S. Lee, J. Bang, and G. Kim, "Moth detection from pheromone trap images using deep learning object detectors," *Agriculture*, vol. 10, no. 5, p. 170, 2020.

[33] J. Wang, C. Lin, L. Ji, and A. Liang, "A new automatic identification system of insect images at the order level," *Knowledge-Based Systems*, vol. 33, pp. 102–110, 2012.

[34] T. Liu, W. Chen, W. Wu, C. Sun, W. Guo, and X. Zhu, "Detection of aphids in wheat fields using a computer vision technique," *Biosystems Engineering*, vol. 141, pp. 82–93, 2016.

[35] Y. Qing, L. Jun, Q.-j. Liu, G.-q. Diao, B.-j. Yang, H.-m. Chen, and T. Jian, "An insect imaging system to automate rice light-trap pest identification," *Journal of Integrative Agriculture*, vol. 11, no. 6, pp. 978–985, 2012.

[36] W. Ding and G. Taylor, "Automatic moth detection from trap images for pest management," *Computers and Electronics in Agriculture*, vol. 123, pp. 17–28, 2016.

[37] C. Xie, R. Wang, J. Zhang, P. Chen, W. Dong, R. Li, T. Chen, and H. Chen, "Multi-level learning features for automatic classification of field crop pests," *Computers and Electronics in Agriculture*, vol. 152, pp. 233–241, 2018.

[38] P. Rajan, B. Radhakrishnan, and L. P. Suresh, "Detection and classification of pests from crop images using support vector machine," in *2016 international conference on emerging technological trends (ICETT)*. IEEE, 2016, pp. 1–6.

[39] E. Omrani, B. Khoshnevisan, S. Shamshirband, H. Saboohi, N. B. Anuar, and M. H. N. M. Nasir, "Potential of radial basis function-based support vector regression for apple disease detection," *Measurement*, vol. 55, pp. 512–519, 2014.

[40] Y. Kaya and L. Kayci, "Application of artificial neural network for automatic detection of butterfly species using color and texture features," *The visual computer*, vol. 30, pp. 71–79, 2014.

[41] K. Asefpour Vakilian and J. Massah, "Performance evaluation of a machine vision system for insect pests identification of field crops using artificial neural networks," *Archives of Phytopathology and plant protection*, vol. 46, no. 11, pp. 1262–1269, 2013.

[42] R. Samanta and I. Ghosh, "Tea insect pests classification based on artificial neural networks," *International Journal of Computer Engineering Science (IJCES)*, vol. 2, no. 6, pp. 1–13, 2012.

[43] K. Espinoza, D. L. Valera, J. A. Torres, A. López, and F. D. Molina-Aiz, "Combination of image processing and artificial neural networks as a novel approach for the identification of bemisia tabaci and frankliniella occidentalis on sticky traps in greenhouse agriculture," *Computers and Electronics in Agriculture*, vol. 127, pp. 495–505, 2016.

[44] B. Qiao, C. Li, V. W. Allen, M. Shirasu-Hiza, and S. Syed, "Automated analysis of long-term grooming behavior in drosophila using ak-nearest neighbors classifier," *Elife*, vol. 7, p. e34497, 2018.

[45] F. Como, E. Carnesecchi, S. Volani, J. L. Dorne, J. Richardson, A. Bassan, M. Pavan, and E. Benfenati, "Predicting acute contact toxicity of pesticides in honeybees (apis mellifera) through a k-nearest neighbor model," *Chemosphere*, vol. 166, pp. 438–444, 2017.

[46] R. I. Hasan, S. M. Yusuf, and L. Alzubaidi, "Review of the state of the art of deep learning for plant diseases: A broad analysis and discussion," *Plants*, vol. 9, no. 10, p. 1302, 2020.

[47] Z. Gao, Z. Luo, W. Zhang, Z. Lv, and Y. Xu, "Deep learning application in plant stress imaging: a review," *AgriEngineering*, vol. 2, no. 3, p. 29, 2020.

[48] N. T. Nam and P. D. Hung, "Pest detection on traps using deep convolutional neural networks," in *Proceedings of the 1st International Conference on Control and Computer Vision*, 2018, pp. 33–38.

[49] J. G. A. Barbedo and G. B. Castro, "A study on cnn-based detection of psyllids in sticky traps using multiple image data sources," *AI*, vol. 1, no. 2, pp. 198–208, 2020.

[50] D. Patel and N. Bhatt, "Improved accuracy of pest detection using augmentation approach with faster r-cnn," in *IOP Conference Series: Materials Science and Engineering*, vol. 1042, no. 1. IOP Publishing, 2021, p. 012020.

[51] E. L. Mique Jr and T. D. Palaoag, "Rice pest and disease detection using convolutional neural network," in *Proceedings of the 1st international conference on information science and systems*, 2018, pp. 147–151.

[52] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Region-based convolutional networks for accurate object detection and segmentation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 1, pp. 142–158, 2015.

[53] J. R. Uijlings, K. E. Van De Sande, T. Gevers, and A. W. Smeulders, "Selective search for object recognition," *International journal of computer vision*, vol. 104, pp. 154–171, 2013.

[54] L. Tessaro, C. Raffaldi, M. Rossi, and D. Brunelli, "Lightweight synchronization algorithm with self-calibration for industrial lora sensor networks," in *2018 Workshop on Metrology for Industry 4.0 and IoT*. IEEE, 2018, pp. 259–263.

[55] L. A. Libutti, F. D. Igual, L. Pinuel, L. De Giusti, and M. Naiouf, "Benchmarking performance and power of usb accelerators for inference with mlperf," in *Proc. 2nd Workshop Accelerated Mach. Learn.(AccML)*, 2020, pp. 1–15.

[56] M. Antonini, T. H. Vu, C. Min, A. Montanari, A. Mathur, and F. Kawsar, "Resource characterisation of personal-scale sensing models on edge accelerators," in *Proceedings of the First International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, 2019, pp. 49–55.

[57] K. Overton, J. L. Maino, R. Day, P. A. Umina, B. Bett, D. Carnovale, S. Ekesi, R. Meagher, and O. L. Reynolds, "Global crop impacts, yield losses and action thresholds for fall armyworm (spodoptera frugiperda): A review," *Crop Protection*, vol. 145, p. 105641, 2021.

[58] C. Roubal and J. Rouzet, "Development and use of a forecasting model for cydia pomonella," *EPPO Bulletin*, vol. 33, no. 3, pp. 403–405, 2003.

[59] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[60] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[61] D. H. Fisher, "Knowledge acquisition via incremental conceptual clustering," *Machine learning*, vol. 2, pp. 139–172, 1987.

[62] A. Li, B. Zheng, G. Pekhimenko, and F. Long, "Automatic horizontal fusion for gpu kernels," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022, pp. 14–27.

[63] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. pmlr, 2015, pp. 448–456.

[64] M. B. Bejiga, A. Zeggada, A. Nouffidj, and F. Melgani, "A convolutional neural network approach for assisting avalanche search and rescue operations with uav imagery," *Remote Sensing*, vol. 9, no. 2, 2017. [Online]. Available: https://www.mdpi.com/2072-4292/9/2/100

[65] E. Lygouras, N. Santavas, A. Taitzoglou, K. Tarchanidis, A. Mitropoulos, and A. Gasteratos, "Unsupervised human detection with an embedded vision system on a fully autonomous uav for search and rescue operations," *Sensors*, vol. 19, no. 16, 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/16/3542

[66] M.-B. Leduc and A. J. Knudby, "Mapping wild leek through the forest canopy using a uav," *Remote Sensing*, vol. 10, no. 1, 2018. [Online]. Available: https://www.mdpi.com/2072-4292/10/1/70

[67] Y. Zhang, Y. Zhang, and Z. Yu, "A solution for searching and monitoring forest fires based on multiple uavs," in *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2019, pp. 661–666.

[68] Y. Angel, M. Morton, Y. Malbeteau, S. S. C. Negrão, G. M. Fiene, M. A. A. Mousa, M. Tester, and M. McCabe, "Multitemporal Monitoring of Phenotypic Traits in Wild Tomato Species (S. pimpinellifolium) Using UAV-based Hyperspectral Imagery," in *AGU Fall Meeting Abstracts*, vol. 2019, Dec. 2019, pp. B31K–2415.

[69] F. Vanegas, D. Bratanov, K. Powell, J. Weiss, and F. Gonzalez, "A novel methodology for improving plant pest surveillance in vineyards and crops using uav-based

hyperspectral and spatial data," *Sensors*, vol. 18, no. 1, 2018. [Online]. Available: https://www.mdpi.com/1424-8220/18/1/260

[70] J. Kaivosoja, J. Hautsalo, J. Heikkinen, L. Hiltunen, P. Ruuttunen, R. Näsi, O. Niemeläinen, M. Lemsalu, E. Honkavaara, and J. Salonen, "Reference measurements in developing uav systems for detecting pests, weeds, and diseases," *Remote Sensing*, vol. 13, no. 7, 2021. [Online]. Available: https://www.mdpi.com/2072-4292/13/7/1238

[71] E. C. Tetila, B. B. Machado, G. Astolfi, N. A. de Souza Belete, W. P. Amorim, A. R. Roel, and H. Pistori, "Detection and classification of soybean pests using deep learning with uav images," *Computers and Electronics in Agriculture*, vol. 179, p. 105836, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016816991831055X

[72] F. Qi, X. Zhu, G. Mang, M. Kadoch, and W. Li, "Uav network and iot in the sky for future smart cities," *IEEE Network*, vol. 33, no. 2, pp. 96–101, March 2019.

[73] A. Tiurlikova, N. Stepanov, and K. Mikhaylov, "Improving the energy efficiency of a lorawan by a uav-based gateway," in *2019 11th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, Oct 2019, pp. 1–6.

[74] R. A. Nazib and S. Moh, "Energy-efficient and fast data collection in uav-aided wireless sensor networks for hilly terrains," *IEEE Access*, vol. 9, pp. 23 168–23 190, 2021.

[75] R. Vijayanandh, J. Darshan Kumar, M. Senthil Kumar, L. Ahilla Bharathy, and G. Raj Kumar, "Design and fabrication of solar powered unmanned aerial vehicle for border surveillance," in *Proceedings of International Conference on Remote Sensing for Disaster Management*, P. J. Rao, K. N. Rao, and S. Kubo, Eds. Cham: Springer International Publishing, 2019, pp. 61–71.

[76] M. L. Laouira, A. Abdelli, J. B. Othman, and H. Kim, "An efficient wsn based solution for border surveillance," *IEEE*

Transactions on Sustainable Computing, vol. 6, no. 1, pp. 54–65, Jan 2021.

[77] F. Granelli, C. Sacchi, R. Bassoli, R. Cohen, and I. Ashkenazi, "A dynamic and flexible architecture based on uavs for border security and safety," in *Advanced Technologies for Security Applications*, C. Palestini, Ed. Dordrecht: Springer Netherlands, 2020, pp. 295–306.

[78] N. Kalatzis, M. Avgeris, D. Dechouniotis, K. Papadakis-Vlachopapadopoulos, I. Roussaki, and S. Papavassiliou, "Edge computing in iot ecosystems for uav-enabled early fire detection," in *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*, June 2018, pp. 106–114.

[79] A. D. Boursianis, M. S. Papadopoulou, P. Diamantoulakis, A. Liopa-Tsakalidi, P. Barouchas, G. Salahas, G. Karagiannidis, S. Wan, and S. K. Goudos, "Internet of things (iot) and agricultural unmanned aerial vehicles (uavs) in smart farming: A comprehensive review," *Internet of Things*, p. 100187, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660520300238

[80] N. Zhao, W. Lu, M. Sheng, Y. Chen, J. Tang, F. R. Yu, and K.-K. Wong, "Uav-assisted emergency networks in disasters," *IEEE Wireless Communications*, vol. 26, no. 1, pp. 45–51, February 2019.

[81] P. Radoglou-Grammatikis, P. Sarigiannidis, T. Lagkas, and I. Moscholios, "A compilation of uav applications for precision agriculture," *Computer Networks*, vol. 172, p. 107148, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S138912862030116X

[82] N. Ayyappaa, A. Y. Raj, A. Adithya, R. Murali, and A. Vinodh, "Autonomous drone for efficacious blood conveyance," in *2019 4th International Conference on Robotics and Automation Engineering (ICRAE)*, Nov 2019, pp. 99–103.

[83] B. D. Song, K. Park, and J. Kim, "Persistent uav delivery logistics: Milp formulation and efficient heuristic," *Computers & Industrial Engineering*, vol. 120, pp. 418–428,

2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360835218302146

[84] D. Falanga, A. Zanchettin, A. Simovic, J. Delmerico, and D. Scaramuzza, "Vision-based autonomous quadrotor landing on a moving platform," in *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, Oct 2017, pp. 200–207.

[85] F. Sadeghi and S. Levine, "Cad2rl: Real single-image flight without a single real image," 2017.

[86] D. K. Kim and T. Chen, "Deep neural network for real-time autonomous indoor navigation," 2015.

[87] P. Santana, L. Correia, R. Mendonça, N. Alves, and J. Barata, "Tracking natural trails with swarm-based visual saliency," *J. Field Robot.*, vol. 30, no. 1, p. 64–86, Jan. 2013. [Online]. Available: https://doi.org/10.1002/rob.21423

[88] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec 2017.

[89] A. Carrio, C. Sampedro, A. Rodriguez-Ramos, and P. Campoy, "A review of deep learning methods and applications for unmanned aerial vehicles," *Journal of Sensors*, vol. 2017, p. 3296874, Aug 2017. [Online]. Available: https://doi.org/10.1155/2017/3296874

[90] P. Zhu, L. Wen, X. Bian, H. Ling, and Q. Hu, "Vision meets drones: A challenge," 2018.

[91] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ser. MCC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 13–16. [Online]. Available: https://doi.org/10.1145/2342509.2342513

[92] N. Mohamed, J. Al-Jaroodi, I. Jawhar, H. Noura, and S. Mahmoud, "Uavfog: A uav-based fog computing for internet of things," in *2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, Aug 2017, pp. 1–8.

[93] M. Chen, Y. Hao, Y. Li, C.-F. Lai, and D. Wu, "On the computation offloading at ad hoc cloudlet: architecture and service modes," *IEEE Communications Magazine*, vol. 53, no. 6, pp. 18–24, June 2015.

[94] G. Mohanarajah, D. Hunziker, R. D'Andrea, and M. Waibel, "Rapyuta: A cloud robotics platform," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 2, pp. 481–493, April 2015.

[95] M. Mukherjee, V. Kumar, A. Lat, M. Guo, R. Matam, and Y. Lv, "Distributed deep learning-based task offloading for uav-enabled mobile edge computing," in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, July 2020, pp. 1208–1212.

[96] D. Callegaro and M. Levorato, "Optimal edge computing for infrastructure-assisted uav systems," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 2, pp. 1782–1792, 2021.

[97] S. Mahmoud and N. Mohamed, "Broker architecture for collaborative uavs cloud computing," in *2015 International Conference on Collaboration Technologies and Systems (CTS)*, 2015, pp. 212–219.

[98] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, Jan 2017.

[99] E. Petritoli, F. Leccese, and M. Leccisi, "Inertial navigation systems for uav: Uncertainty and error measurements," in *2019 IEEE 5th International Workshop on Metrology for AeroSpace (MetroAeroSpace)*, 2019, pp. 1–5.

[100] O. J. Woodman, "An introduction to inertial navigation," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-696, Aug. 2007. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-696.pdf

[101] B. W. PARKINSON, T. STANSELL, R. BEARD, and K. GROMOV, "A history of satellite navigation," *NAVIGATION*, vol. 42, no. 1, pp. 109–164, 1995. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.2161-4296.1995.tb02333.x

[102] M. H. F. M. Fauadi, S. Akmal, M. M. Ali, N. I. Anuar, S. Ramlan, A. Z. M. Noor, and N. Awang, "Intelligent vision-based navigation system for mobile robot: A technological review," *Periodicals of Engineering and Natural Sciences*, vol. 6, no. 2, pp. 47–57, 2018.

[103] J. Courbon, Y. Mezouar, N. Guénard, and P. Martinet, "Vision-based navigation of unmanned aerial vehicles," *Control Engineering Practice*, vol. 18, no. 7, pp. 789–799, 2010, special Issue on Aerial Robotics. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0967066110000808

[104] S. Li, M. M. Ozo, C. De Wagter, and G. C. de Croon, "Autonomous drone race: A computationally efficient vision-based navigation and control strategy," *Robotics and Autonomous Systems*, vol. 133, p. 103621, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889020304619

[105] S. Y. Choi and D. Cha, "Unmanned aerial vehicles using machine learning for autonomous flight; state-of-the-art," *Advanced Robotics*, vol. 33, no. 6, pp. 265–277, 2019.

[106] M. Paszkuta, J. Rosner, D. Peszor, M. Szender, M. Wojciechowska, K. Wojciechowski, and J. P. Nowacki, "Uav onboard emergency safe landing spot detection system combining classical and deep learning-based segmentation methods," in *Asian Conference on Intelligent Information and Database Systems*. Springer, 2021, pp. 467–478.

[107] P. Mathur, Y. Jangir, and N. Goveas, "A generalized kalman filter augmented deep-learning based approach for autonomous landing in mavs," in *2021 International Symposium of Asian Control Association on Intelligent Robotics and Industrial Automation (IRIA)*. IEEE, 2021, pp. 1–6.

[108] B. H. Y. Alsalam, K. Morton, D. Campbell, and F. Gonzalez, "Autonomous uav with vision based on-board decision making for remote sensing and precision agriculture," in *2017 IEEE Aerospace Conference*. IEEE, 2017, pp. 1–12.

[109] A. Koubâa and B. Qureshi, "Dronetrack: Cloud-based real-time object tracking using unmanned aerial vehicles over the internet," *IEEE Access*, vol. 6, pp. 13 810–13 824, 2018.

[110] J. Lee, J. Wang, D. Crandall, S. Šabanović, and G. Fox, "Real-time, cloud-based object detection for unmanned aerial vehicles," in *2017 First IEEE International Conference on Robotic Computing (IRC)*. IEEE, 2017, pp. 36–43.

[111] A. Albanese, M. Nardello, and D. Brunelli, "Automated pest detection with dnn on the edge for precision agriculture," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 11, no. 3, pp. 458–467, 2021.

[112] S. Liu and W. Deng, "Very deep convolutional neural network based image classification using small training sample size," in *2015 3rd IAPR Asian conference on pattern recognition (ACPR)*. IEEE, 2015, pp. 730–734.

[113] S. Jung, S. Hwang, H. Shin, and D. H. Shim, "Perception, guidance, and navigation for indoor autonomous drone racing using deep learning," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 2539–2544, 2018.

[114] X. Xie, X. Han, Q. Liao, and G. Shi, "Visualization and pruning of ssd with the base network vgg16," in *Proceedings of the 2017 International Conference on Deep Learning Technologies*, 2017, pp. 90–94.

[115] S.-H. Lee, C.-H. Yeh, T.-W. Hou, and C.-S. Yang, "A lightweight neural network based on alexnet-ssd model for

garbage detection," in *Proceedings of the 2019 3rd High Performance Computing and Cluster Technologies Conference*, 2019, pp. 274–278.

[116] S. Gu, L. Ding, Y. Yang, and X. Chen, "A new deep learning method based on alexnet model and ssd model for tennis ball recognition," in *2017 IEEE 10th International Workshop on Computational Intelligence and Applications (IWCIA)*. IEEE, 2017, pp. 159–164.

[117] N. Tijtgat, W. Van Ranst, T. Goedeme, B. Volckaert, and F. De Turck, "Embedded real-time object detection for a uav warning system," in *Proceedings of the IEEE International Conference on Computer Vision Workshops*, 2017, pp. 2110–2118.

[118] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263–7271.

[119] I. Khokhlov, E. Davydenko, I. Osokin, I. Ryakin, A. Babaev, V. Litvinenko, and R. Gorbachev, "Tiny-yolo object detection supplemented with geometrical data," in *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*. IEEE, 2020, pp. 1–5.

[120] R. Mishra, H. P. Gupta, and T. Dutta, "A survey on deep neural network compression: Challenges, overview, and solutions," *arXiv preprint arXiv:2010.03954*, 2020.

[121] S. Ye, X. Feng, T. Zhang, X. Ma, S. Lin, Z. Li, K. Xu, W. Wen, S. Liu, J. Tang *et al.*, "Progressive dnn compression: A key to achieve ultra-high weight pruning and quantization rates using admm," *arXiv preprint arXiv:1903.09769*, 2019.

[122] E.-H. Yang, H. Amer, and Y. Jiang, "Compression helps deep learning in image classification," *Entropy*, vol. 23, no. 7, p. 881, 2021.

[123] Z. Hu, X. Zou, W. Xia, S. Jin, D. Tao, Y. Liu, W. Zhang, and Z. Zhang, "Delta-dnn: Efficiently compressing deep neural networks via exploiting floats similarity," in *49th International Conference on Parallel Processing-ICPP*, 2020, pp. 1–12.

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

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

[126] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[127] C. Kyrkou, G. Plastiras, T. Theocharides, S. I. Venieris, and C.-S. Bouganis, "Dronet: Efficient convolutional neural network detector for real-time uav applications," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 967–972.

[128] G. Plastiras, C. Kyrkou, and T. Theocharides, "Edgenet: Balancing accuracy and performance for edge-based convolutional neural network object detectors," in *Proceedings of the 13th International Conference on Distributed Smart Cameras*, 2019, pp. 1–6.

[129] R. Wu, X. Guo, J. Du, and J. Li, "Accelerating neural network inference on fpga-based platforms—a survey," *Electronics*, vol. 10, no. 9, p. 1025, 2021.

[130] M. Murshed, C. Murphy, D. Hou, N. Khan, G. Ananthanarayanan, and F. Hussain, "Machine learning at the network edge: A survey," *arXiv preprint arXiv:1908.00080*, 2019.

[131] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[132] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[133] F. Argenteri, "Enhancing uav capabilities with machine learning on board," 2020. [Online]. Available: https://github.com/frank1789/MasterThesis

[134] L. R. Agostinho, N. M. Ricardo, M. I. Pereira, A. Hiolle, and A. M. Pinto, "A practical survey on visual odometry for autonomous driving in challenging scenarios and conditions," *IEEE Access*, vol. 10, pp. 72 182–72 205, 2022.

[135] Y. Zhang, A. Carballo, H. Yang, and K. Takeda, "Perception and sensing for autonomous vehicles under adverse weather conditions: A survey," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 196, pp. 146–177, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0924271622003367

[136] Y. X. Tan, M. Meghjani, and M. B. Prasetyo, "Localization with anticipation for autonomous urban driving in rain," *arXiv preprint arXiv:2306.09134*, 2023.

[137] S.-H. Wang, S.-C. Hsia, and M.-J. Zheng, "Deep learning-based raindrop quantity detection for real-time vehicle-safety application," *IEEE Transactions on Consumer Electronics*, vol. 67, no. 4, pp. 266–274, 2021.

[138] C. P. T. Kondapalli, V. Vaibhav, K. R. Konda, K. Praveen, and B. Kondoju, "Real-time rain severity detection for autonomous driving applications," in *2021 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2021, pp. 1451–1456.

[139] P. McEnroe, S. Wang, and M. Liyanage, "A survey on the convergence of edge computing and ai for uavs: Opportunities and challenges," *IEEE Internet of Things Journal*, vol. 9, no. 17, pp. 15 435–15 459, 2022.

[140] Z. Wei, M. Zhu, N. Zhang, L. Wang, Y. Zou, Z. Meng, H. Wu, and Z. Feng, "Uav-assisted data collection for internet of things: A survey," *IEEE Internet of Things Journal*, vol. 9, no. 17, pp. 15 460–15 483, 2022.

[141] Y. Wang, Z. Gao, J. Zhang, X. Cao, D. Zheng, Y. Gao, D. W. K. Ng, and M. D. Renzo, "Trajectory design for uav-based internet of things data collection: A deep reinforcement

learning approach," *IEEE Internet of Things Journal*, vol. 9, no. 5, pp. 3899–3912, 2022.

[142] R. Han, J. Wang, L. Bai, J. Liu, and J. Choi, "Age of information and performance analysis for uav-aided iot systems," *IEEE Internet of Things Journal*, vol. 8, no. 19, pp. 14 447–14 457, 2021.

[143] A. Albanese, T. Taccioli, T. Apicella, D. Brunelli, and E. Ragusa, "Design and deployment of an efficient landing pad detector," in *International Conference on System-Integrated Intelligence*. Springer, 2022, pp. 137–147.

[144] L. Santoro, A. Albanese, M. Canova, M. Rossa, D. Fontanelli, and D. Brunelli, "A plug-and-play tinyml-based vision system for drone automatic landing," in *2023 IEEE International Workshop on Metrology for Industry 4.0 & IoT (MetroInd4.0&IoT)*. IEEE, 2023, pp. 293–298.

[145] M. Gao, C. H. Hugenholtz, T. A. Fox, M. Kucharczyk, T. E. Barchyn, and P. R. Nesbit, "Weather constraints on global drone flyability," *Scientific reports*, vol. 11, no. 1, p. 12092, 2021.

[146] M. Eskandari and A. V. Savkin, "Deep-reinforcement-learning-based joint 3-d navigation and phase-shift control for mobile internet of vehicles assisted by ris-equipped uavs," *IEEE Internet of Things Journal*, vol. 10, no. 20, pp. 18 054–18 066, 2023.

[147] C. Wang, J. Wang, J. Wang, and X. Zhang, "Deep-reinforcement-learning-based autonomous uav navigation with sparse rewards," *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 6180–6190, 2020.

[148] W. Maddern, G. Pascoe, C. Linegar, and P. Newman, "1 Year, 1000km: The Oxford RobotCar Dataset," *The International Journal of Robotics Research (IJRR)*, vol. 36, no. 1, pp. 3–15, 2017. [Online]. Available: http://dx.doi.org/10.1177/0278364916679498

[149] W. Maddern, G. Pascoe, M. Gadd, D. Barnes, B. Yeomans, and P. Newman, "Real-time kinematic ground truth for the

oxford robotcar dataset," *arXiv preprint arXiv: 2002.10152*, 2020. [Online]. Available: https://arxiv.org/pdf/2002.10152

[150] I. Fursa, E. Fandi, V. Musat, J. Culley, E. Gil, I. Teeti, L. Bilous, I. V. Sluis, A. Rast, and A. Bradley, "Worsening perception: Real-time degradation of autonomous vehicle perception performance for simulation of adverse weather conditions," *arXiv preprint arXiv:2103.02760*, 2021.

[151] M. Hnewa and H. Radha, "Object detection under rainy conditions for autonomous vehicles: A review of state-of-the-art and emerging techniques," *IEEE Signal Processing Magazine*, vol. 38, no. 1, pp. 53–67, 2020.

[152] E. O. Appiah and S. Mensah, "Object detection in adverse weather condition for autonomous vehicles," *Multimedia Tools and Applications*, vol. 83, no. 9, pp. 28 235–28 261, 2024.

[153] Y. Wang and D. Boyle, "Trustworthy reinforcement learning for quadrotor uav tracking control systems," *arXiv preprint arXiv:2302.11694*, 2023.

[154] Y. Wang, J. O'Keeffe, Q. Qian, and D. Boyle, "Kinojgm: A framework for efficient and accurate quadrotor trajectory generation and tracking in dynamic environments," in *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 2022, pp. 11 036–11 043.

[155] Y. Wang, Q. Qian, and D. Boyle, "Probabilistic constrained reinforcement learning with formal interpretability," in *International Conference on Machine Learning*. PMLR, 2024.

[156] T. Brophy, D. Mullins, A. Parsi, J. Horgan, E. Ward, P. Denny, C. Eising, B. Deegan, M. Glavin, and E. Jones, "A review of the impact of rain on camera-based perception in automated driving systems," *IEEE Access*, 2023.

[157] Z. Zhang, Y. Wei, H. Zhang, Y. Yang, S. Yan, and M. Wang, "Data-driven single image deraining: A comprehensive review and new perspectives," *Pattern Recognition*, p. 109740, 2023.

[158] L. Yan, W. Zai, J. Wang, and D. Yang, "Image defogging method for transmission channel inspection by uav based on

deep multi-patch layered network," in *2023 Panda Forum on Power and Energy (PandaFPE)*. IEEE, 2023, pp. 855–860.

[159] F. Zhang, S. You, Y. Li, and Y. Fu, "Gtav-nightrain: Photometric realistic large-scale dataset for night-time rain streak removal," *arXiv preprint arXiv:2210.04708*, 2022.

[160] S.-C. Huang, Q.-V. Hoang, and T.-H. Le, "Sfa-net: A selective features absorption network for object detection in rainy weather conditions," *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[161] T. Qin, P. Li, and S. Shen, "Vins-mono: A robust and versatile monocular visual-inertial state estimator," *IEEE Transactions on Robotics*, vol. 34, no. 4, pp. 1004–1020, 2018.

[162] T. Qin and S. Shen, "Online temporal calibration for monocular visual-inertial systems," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 3662–3669.

[163] T. Qin, J. Pan, S. Cao, and S. Shen, "A general optimization-based framework for local odometry estimation with multiple sensors," 2019.

[164] T. Qin, S. Cao, J. Pan, and S. Shen, "A general optimization-based framework for global pose estimation with multiple sensors," 2019.

[165] S. Hasirlioglu and A. Riener, "A general approach for simulating rain effects on sensor data in real and virtual environments," *IEEE Transactions on Intelligent Vehicles*, vol. 5, no. 3, pp. 426–438, 2019.

[166] A. Albanese, M. Nardello, G. Fiacco, and D. Brunelli, "Tiny machine learning for high accuracy product quality inspection," *IEEE Sensors Journal*, vol. 23, no. 2, pp. 1575–1583, 2022.

[167] A. Albanese and D. Brunelli, "Industrial visual inspection with tinyml for high-performance quality control," *IEEE Instrumentation & Measurement Magazine*, vol. 26, no. 8, pp. 17–22, 2023.

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

[169] H. Chopra, H. Singh, M. S. Bamrah, F. Mahbubani, A. Verma, N. Hooda, P. S. Rana, R. K. Singla, and A. K. Singh, "Efficient fruit grading system using spectrophotometry and machine learning approaches," *IEEE Sensors Journal*, vol. 21, no. 14, pp. 16 162–16 169, 2021.

[170] J. Tao, Y. Zhu, F. Jiang, H. Liu, and H. Liu, "Rolling surface defect inspection for drum-shaped rollers based on deep learning," *IEEE Sensors Journal*, vol. 22, no. 9, pp. 8693–8700, 2022.

[171] Z. Wang, J. Bai, X. Zhang, X. Qin, X. Tan, and Y. Zhao, "Base detection research of drilling robot system by using visual inspection," *Journal of Robotics*, vol. 2018, 2018.

[172] D. L. Dutta and S. Bharali, "Tinyml meets IoT: A comprehensive survey," *Internet of Things*, vol. 16, p. 100461, 2021.

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

[174] Y. Zou, M. Gottardi, M. Lecca, and M. Perenzoni, "A low-power vga vision sensor with embedded event detection for outdoor edge applications," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 11, pp. 3112–3121, 2020.

[175] L. Tsutsui da Silva, V. M. A. Souza, and G. E. A. P. A. Batista, "An open-source tool for classification models in resource-constrained hardware," *IEEE Sensors Journal*, vol. 22, no. 1, pp. 544–554, 2022.

[176] J. Schmitt, J. Bönig, T. Borggräfe, G. Beitinger, and J. Deuse, "Predictive model-based quality inspection using machine learning and edge cloud computing," *Advanced engineering informatics*, vol. 45, p. 101101, 2020.

[177] M. A. Ali and A. K. Lun, "A cascading fuzzy logic with image processing algorithm–based defect detection for automatic visual inspection of industrial cylindrical object's surface," *The International Journal of Advanced Manufacturing Technology*, vol. 102, no. 1, pp. 81–94, 2019.

[178] X. Zhou, Y. Wang, Q. Zhu, J. Mao, C. Xiao, X. Lu, and H. Zhang, "A surface defect detection framework for glass bottle bottom using visual attention model and wavelet transform," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 4, pp. 2189–2201, 2020.

[179] S. Unnikrishnan, J. Donovan, R. Macpherson, and D. Tormey, "An integrated histogram-based vision and machine-learning classification model for industrial emulsion processing," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 9, pp. 5948–5955, 2020.

[180] C. Ge, J. Wang, J. Wang, Q. Qi, H. Sun, and J. Liao, "Towards automatic visual inspection: A weakly supervised learning method for industrial applicable object detection," *Computers in Industry*, vol. 121, p. 103232, 2020.

[181] P. Ren, Y. Xiao, X. Chang, P.-y. Huang, Z. Li, X. Chen, and X. Wang, "A comprehensive survey of neural architecture search: Challenges and solutions," *ACM Comput. Surv.*, vol. 54, no. 4, may 2021.

[182] A. Imteaj, U. Thakker, S. Wang, J. Li, and M. H. Amini, "A survey on federated learning for resource-constrained iot devices," *IEEE Internet of Things Journal*, vol. 9, no. 1, pp. 1–24, 2022.

[183] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, "Mcunet: Tiny deep learning on iot devices," *arXiv preprint arXiv:2007.10319*, 2020.

[184] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*. PMLR, 2019, pp. 6105–6114.

[185] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han, "Mcunetv2: Memory-efficient patch-based inference for tiny deep learn-

ing," in *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[186] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[187] H. Ren, D. Anicic, and T. Runkler, "The synergy of complex event processing and tiny machine learning in industrial iot," *arXiv preprint arXiv:2105.03371*, 2021.

[188] A. Mostafavi and A. Sadighi, "A novel online machine learning approach for real-time condition monitoring of rotating machines," in *2021 9th RSI International Conference on Robotics and Mechatronics (ICRoM)*. IEEE, 2021, pp. 267–273.

[189] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[190] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint:1510.00149*, 2015.

[191] B. Sudharsan, P. Yadav, J. G. Breslin, and M. I. Ali, "An sram optimized approach for constant memory consumption and ultra-fast execution of ml classifiers on tinyml hardware," in *2021 IEEE International Conference on Services Computing (SCC)*. IEEE, 2021, pp. 319–328.

[192] H. Cai, C. Gan, L. Zhu, and S. Han, "Tinytl: Reduce memory, not parameters for efficient on-device learning," *arXiv preprint:2007.11622*, 2020.

[193] J. Schwarz, W. Czarnecki, J. Luketina, A. Grabska-Barwinska, Y. W. Teh, R. Pascanu, and R. Hadsell, "Progress & compress: A scalable framework for continual learning," in *International Conference on Machine Learning*. PMLR, 2018, pp. 4528–4537.

[194] H. Ren, D. Anicic, and T. Runkler, "Tinyol: Tinyml with online-learning on microcontrollers," *arXiv preprint arXiv:2103.08295*, 2021.

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

[196] G.-M. Park, S.-M. Yoo, and J.-H. Kim, "Convolutional neural network with developmental memory for continual learning," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[197] S. Disabato and M. Roveri, "Incremental on-device tiny machine learning," in *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, 2020, pp. 7–13.

[198] A. Zhou, R. Muller, and J. Rabaey, "Memory-efficient, limb position-aware hand gesture recognition using hyperdimensional computing," *arXiv preprint:2103.05267*, 2021.

[199] M. M. Grau, R. P. Centelles, and F. Freitag, "On-device training of machine learning models on microcontrollers with a look at federated learning," in *Proceedings of the Conference on Information Technology for Social Good*, 2021, pp. 198–203.

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

[201] T. Lesort, V. Lomonaco, A. Stoian, D. Maltoni, D. Filliat, and N. Díaz-Rodríguez, "Continual learning for robotics: Definition, framework, learning strategies, opportunities and challenges," *Information fusion*, vol. 58, pp. 52–68, 2020.

[202] Z. Li and D. Hoiem, "Learning without forgetting," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 12, pp. 2935–2947, 2017.

[203] F. Zenke, B. Poole, and S. Ganguli, "Continual learning through synaptic intelligence," in *International Conference on Machine Learning*. PMLR, 2017, pp. 3987–3995.

[204] D. Maltoni and V. Lomonaco, "Continuous learning in single-incremental-task scenarios," *Neural Networks*, vol. 116, pp. 56–73, 2019.

[205] R. M. French, "Catastrophic forgetting in connectionist networks," *Trends in cognitive sciences*, vol. 3, no. 4, pp. 128–135, 1999.

[206] V. Lomonaco and D. Maltoni, "Core50: a new dataset and benchmark for continuous object recognition," in *Conference on Robot Learning*. PMLR, 2017, pp. 17–26.

[207] F. Pilati and A. Sbaragli, "Learning human-process interaction in manual manufacturing job shops through indoor positioning systems," *Computers in Industry*, vol. 151, p. 103984, 2023.

[208] F. Pilati, A. Sbaragli, T. Ruppert, and J. Abonyi, "Goal-oriented clustering algorithm to monitor the efficiency of logistic processes through real-time locating systems," *International Journal of Computer Integrated Manufacturing*, pp. 1–17, 2024.

[209] F. Pilati, A. Sbaragli, G. P. R. Papini, and P. Capuccini, "An artificial neural network architecture to classify workers' operations in manual production processes," in *International Conference on Flexible Automation and Intelligent Manufacturing*. Springer, 2023, pp. 805–812.

[210] X. Zhan, J. Xie, Z. Liu, Y.-S. Ong, and C. C. Loy, "Online deep clustering for unsupervised representation learning," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. IEEE, 2020, pp. 6688–6697.

[211] A. M. N. Taufique, C. S. Jahan, and A. Savakis, "Unsupervised continual learning for gradually varying domains," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 3740–3750.

[212] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han, "On-device training under 256kb memory," *Advances in Neural Information Processing Systems*, vol. 35, pp. 22 941–22 954, 2022.

[213] H. Desai, M. Nardello, D. Brunelli, and B. Lucia, "Camaroptera: A long-range image sensor with local inference for remote sensing applications," *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 3, may 2022.

[214] K. Muhammad, T. Hussain, J. Del Ser, V. Palade, and V. H. C. de Albuquerque, "Deepres: A deep learning-based video summarization strategy for resource-constrained industrial surveillance scenarios," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 9, pp. 5938–5947, 2020.

[215] K. Thandiackal, L. Piccinelli, P. Pati, and O. Goksel, "Multi-scale feature alignment for continual learning of unlabeled domains," *arXiv preprint arXiv:2302.01287*, 2023.