ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF INDUSTRIAL ENGINEER

SECOND CYCLE DEGREE

Deep Learning-Based Spacecraft Detection Algorithm Optimized for Embedded Hardware

Dissertation in Spacecraft Attitude Dynamics and Control

Supervisor
Prof. Dario Modenini
Co-Supervisor:
PhD candidate:
Roman Prokazov

Candidate:
Michele Sacripante

Graduation Session 03/2025
Academic Year 2023/2024

Abstract

Since the launch of the first space mission in 1957, the number of artificial satellites on Low Earth Orbit (LEO) has grown exponentially, leading to a significant technological and scientific advancement. However, the lack of stringent regulations, especially in the early years of space exploration, has also resulted in a substantial accumulation of space debris. The problem of space debris is of great concern to the international scientific community, as it might preclude access to space for future missions and negatively influence existing ones.
For this reason, aerospace companies worldwide are actively developing on-orbit servicing and space debris removal missions. In order for these missions to be successful, highly autonomous navigation systems are essential. Traditional methods, such as those relying on expensive and cumbersome Light Detection and Ranging (LIDAR) systems, are being replaced by vision-based navigation approaches. This shift is driven by the need for cost-effective and adaptable solutions capable of operating in the dynamic and unpredictable space environment. To enable accurate and reliable vision-based navigation, robust object detection capabilities are paramount. Deep Learning (DL) algorithms are increasingly recognized for their potential to achieve the required level of autonomy and accuracy in this critical task.
This thesis focuses on advancing the application of DL for spacecraft object detection by developing and optimizing an image classification network. Specifically, we leveraged the EfficientNet-B0 architecture to create a lightweight yet accurate object detection network tailored for deployment on embedded hardware, such as the Jetson Orin Nano. To further enhance efficiency, we explored quantization techniques, converting the model through Torch, ONNX, and TensorRT formats. Additionally, we trained a YOLO network to provide a comparative benchmark for our developed solution. Both networks were trained using the SPEED+ dataset, a comprehensive resource for spacecraft pose estimation. The primary aim of this work is to demonstrate the feasibility of deploying highly efficient DL-based object detection on resource-constrained platforms, thereby contributing to the development of robust and autonomous space debris removal missions. Indeed, our software pipeline was capable of running at up to 136.05 fps showing satisfactory object detection performance, with an average Intersection over Union index equal to 0.850 on synthetic images, which drops to about 0.382 during domain gap tests.

Acknowledgements

I would like to thank Professor Modenini for his support and for allowing me to participate in this project. I am also grateful to Roman Prokazov for his guidance, valuable advice and availability.

A special thanks goes to my family, who have supported and encouraged me throughout these years at university.

Finally, I want to thank all the friends I have met along the way and who have contributed to this journey with their friendship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The main goal of this thesis is to develop and optimize a real-time vision-based object detector, utilizing deep learning algorithms, specifically for space applications. A crucial aspect of these applications, particularly in scenarios like on-orbit servicing and debris removal, is the ability to rapidly process visual data. Therefore, this work focuses on optimizing the inference speed of the object detector, meaning the time required for the model to generate a prediction. This optimization is essential to address the growing challenge of space debris, a critical issue within the international scientific community.
The organization of this document is as follows. *Chapter 1* presents the main concepts related to the space debris problem and its potential impact on current and future space missions. It is followed by an introduction to Artificial Intelligence (AI) and its increasingly important role in space applications.
Chapter 2 presents the state-of-the-art techniques for building and deploying real-time vision-based object detection systems. In particular, an introduction to the field of Computer Vision (CV) and how this discipline has benefited from AI is provided. Subsequently, the main DL algorithms, learning mechanisms, and evaluation metrics are discussed. Finally, an overview of techniques for optimizing DL inference speed is presented.
Chapter 3 provides a description of the main tools used during the thesis work, with a focus on their application during the training, validation, and deployment of the developed and optimized model on embedded hardware.
In *Chapter 4*, the obtained results for a custom model are presented and discussed. Subsequently, the results are validated by comparing them with those obtained for a YOLOv8n model from Ultralytics and relevant literature.
Finally, in *Chapter 5*, the conclusions of the work are presented, along with possible scenarios for future research.

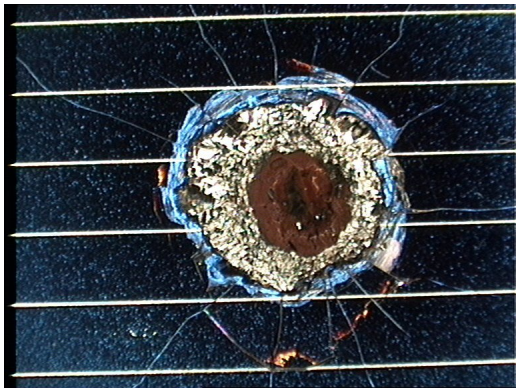
1.1 Space Debris

The *Kessler Syndrome* is a theory that describes a scenario where the density of satellites and space debris in orbit becomes so high that collisions between spacecraft and space debris, and between satellites themselves, are inevitable. These collisions will then lead to more debris release, which will generate further collisions,

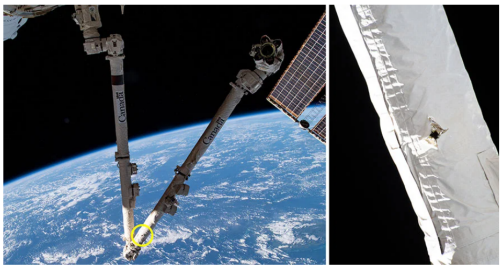
giving way to a cascade reaction [38]. This theory, presented by Donald J. Kessler in 1978, indicates the extent of the space debris problem, which encompasses all non-functional, artificial objects, including fragments and elements thereof, in Earth orbit or re-entering into Earth's atmosphere [3].
Since the onset of the space race, orbits such as LEO, medium Earth orbit (MEO) and, geostationary Earth orbit (GEO) have become increasingly populated by satellites, leading to a dramatic growth in the number of space debris. This proliferation is driven by plenty of sources of space debris such as spent rocket stages, satellite explosions, anti-satellite tests, and in-orbit collisions.
The US Space Surveillance Network is the body responsible for cataloging and tracking objects larger than approximately 5-10 cm in LEO and 30 cm to 1 m at GEO altitudes. As of today, out of the 36860 tracked objects, only about 10200 are intact, operational satellites. In addition, many objects remain untracked and uncatalogued. Statistical models estimate that, in 2024, there were [3]:

- 40500 space debris objects greater than 10 cm;
- 1100000 space debris objects from greater than 1 cm to 10 cm;
- 130 million space debris objects from greater than 1 mm to 1 cm.

The real problem related to space debris is that even debris smaller than 1 cm can cause catastrophic damage due to the high velocities at which they travel. Examples of impacts between satellites and high-speed debris are illustrated in figure 1.1.



(a) Hubble solar cell impact damage [2].



(b) Canadarm2 damage - Image credit NASA/Canadian Space Agency.

Figure 1.1: Examples of damage caused by space debris

Several approaches have been developed to address the problem of space debris, ranging from passive shielding techniques to advanced technologies for identifying, capturing, and removing debris [54].
To perform these operations, it is essential to determine the attitude, geometry, and composition of the debris. Therefore, robust object detection algorithms are required to retrieve this information, enabling accurate maneuver planning for collision avoidance or efficient debris collection while supporting autonomous navigation during capture operations [1].

1.2 Artificial Intelligence, Machine Learning and Deep Learning

Artificial Intelligence is the ability of a machine to show human ability such as reasoning, learning, planning and creativity. AI allows systems to understand their environment, interpret external inputs and provide solutions to problems [53].
The theory of AI was born in 1943, with the work of McCulloch and Pitts, but it is thanks to the technological advances of recent years that it is becoming more and more present in everyday life.
The simplest task that can be performed by AI is commonly referred to as Symbolic AI, where given in input data and rules the machine is able to generate answers.
Machine Learning (ML) is a subset of AI that includes techniques enabling computers to learn how to perform tasks without being explicitly programmed. This process is the result of a training specific, where the model is provided with data and the corresponding outputs, and from there, it infers patterns and develops rules. Once those rules are defined, ML enables the automation of tasks through the generation of adaptive and generalizable rules [53].
Finally *Deep Learning* is a specific subgroup of ML focused on layered techniques inspired by the structure and function of the brain. It is based on Artificial Neural Networks (ANN) in which multiple layers of processing are used to extract progressively higher level features from data [53].
Figure 1.2 gives a visual representation of how the AI could be subdivided.

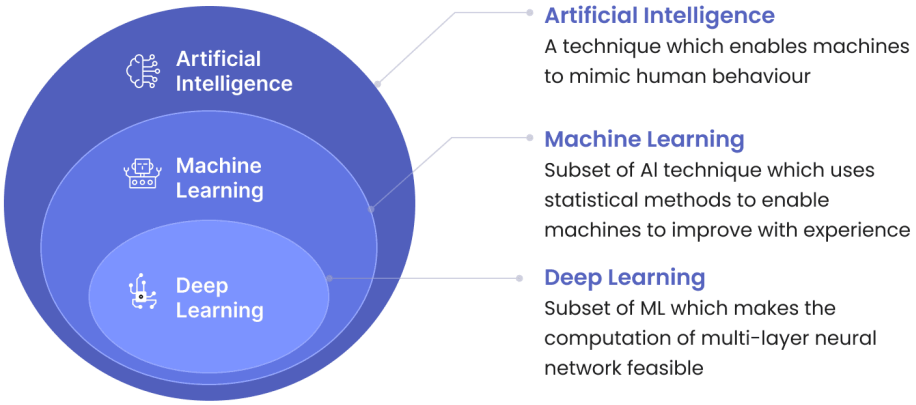
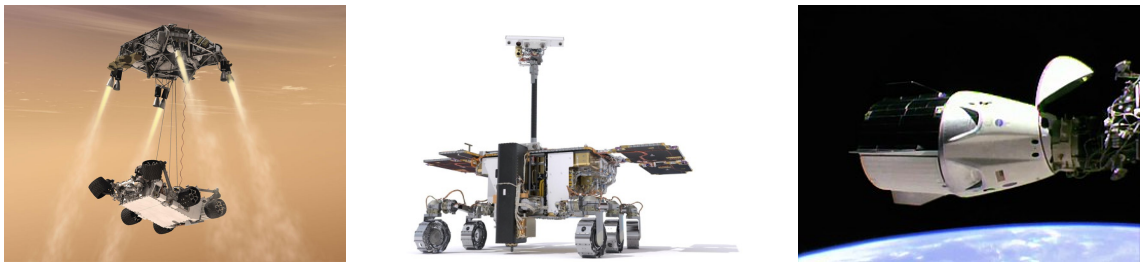


Figure 1.2: Visual division of AI in ML and DL [55]

1.3 Artificial Intelligence for space applications

Today, it is increasingly evident that terrestrial technology is dependent on and related to space technology, as demonstrated, for instance, in the near global internet coverage and GNSS services. As a result, the number of space missions launched every year is growing, leading to, as already mentioned above, an ever-increasing population of the main orbits around Earth.

While there has been a general improvement in the quality of life, there are also new challenges that need to be addressed. Some of them include limited communication windows, long communication latencies, limited bandwidth, restricted access and availability of operators, limited crew availability, system complexity, sudden maneuvers and many other factors often preclude direct human oversight of many functions.
This highlights the need for intelligent systems that can make decisions on their own in remote, potentially hostile environments. AI can then play a fundamental role in the life of a satellite, saving time and money. Space applications of AI can be divided into three types of operations they support: predictable, unpredictable, and real time [36].



(a) Mars 2020 mission [47]

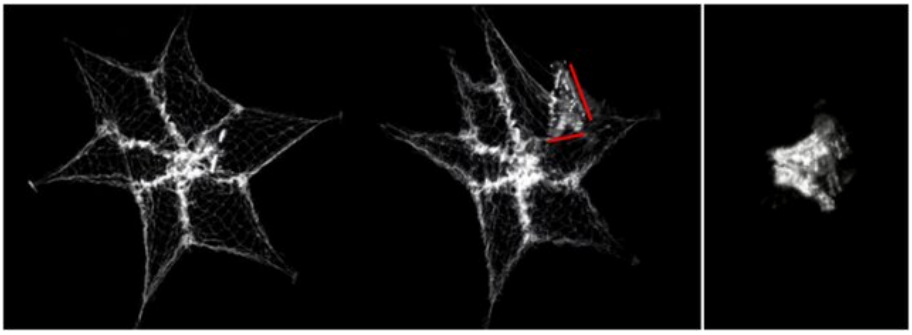
(b) Rosalind Franklin rover from the ExoMars mission [8]

(c) Crew Dragon capsule [34]

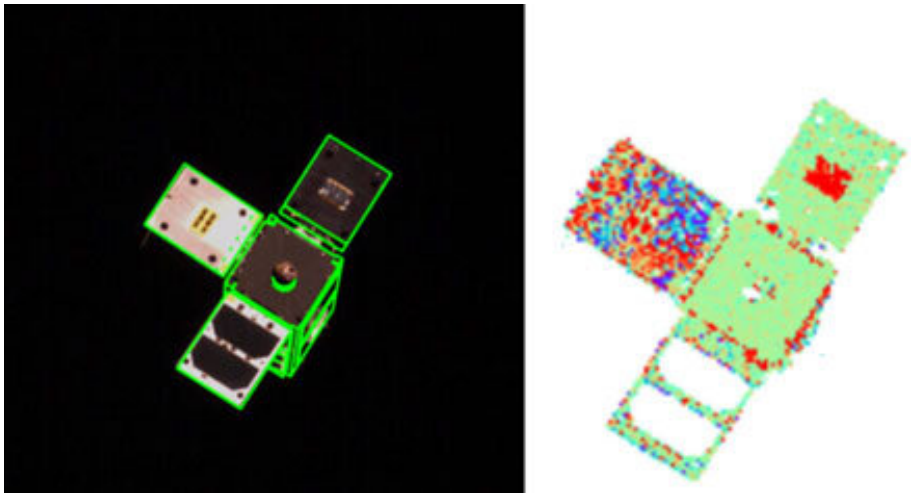
Figure 1.3: Examples of space missions with AI algorithms involvement

Many flight operations such as navigation and maneuvering in space, orbiting a celestial body, observations, communication, and safekeeping activities are highly predictable and can be planned well in advance. The complexity of operations, coupled with stringent resource constraints, underscores the necessity for automated planning and scheduling. An example of a space mission with this kind of AI application is NASA's Mars 2020, that used autonomous navigation algorithm based on the ground conformation (figure 1.3a) [33].
In contrast, surface operations such as long- and short-range traverse, sensing, approaching an object of interest to place tools in contact with it, drilling, coring, sampling, assembly of structures, and many others are characterized by a high degree of uncertainty resulting from interactions with the environment. Operations in these environments, without the autonomy to monitor progress and adjust behavior accordingly, would be greatly restricted, particularly as communication delays to Earth increase. ESA's ExoMars mission leverages AI to autonomously analyze and identify drilled rock samples, enabling the detection of potential organic compounds and advancing our understanding of Mars's geological and possibly biological history (Figure 1.3b) [37].
Finally, operations such as entry, descent, and landing (EDL) and automated spacecraft docking require a real-time response from the vehicle that can preclude any interaction with mission control. As it happens in the capsule Crew Dragon of Space X during autonomous docking with the ISS (figure 1.3c).
A critical factor for real-time applications is the reactivity of the vehicle to prevent

catastrophic outcomes. Faster response times enable proactive decision-making, such as performing collision avoidance manoeuvres and active debris removal.
The application of AI to CV is of particular interest for the development of robust and reliable identification algorithms. These algorithms play a key role in the development of In Orbit Services, sector characterized by both unpredictable and real-time scenarios. One area that has particularly benefited from this technology is active space debris removal, which relies on precise identification and tracking of debris.
A notable example is given from the RemoveDEBRIS mission [29], a technology demonstration mission featuring three main experiments: net capture, harpoon capture, and a vision-based navigation algorithm. Although CubeSats were ejected and used as targets instead of real space debris for the purposes of the mission, it still represents an important step toward a fully operational active debris removal mission.
In this context, the vision-based algorithm detects in real-time the CubeSat and provides information that is later used for pose estimation. Based on this data, a net is deployed to capture the satellite.
Figure 1.4 shows real images from the mission, particularly during the satellite capture with the net and the results of the object detection process.
This, once again, confirms the critical role that AI can play in advancing space technology.



(a) Left -before the capture, two lateral booms visible. Centre -moment of the Net capture of DSAT#1, one of the satellite sails is shown, between the lateral and longitudinal booms. Right -after the capture, DSAT#1 tangled in the net



(b) Left: View of DSAT#2 with shape contours, Right: image from LiDAR camera.

Figure 1.4: Real images from the ESA RemoveDEBRIS mission for active space debris collection [4].

Chapter 2

Theoretical background

This chapter presents the theoretical concepts that support the development of the work presented in this thesis. In particular, it outlines classical CV techniques, highlighting their limitations and how they can be enhanced by AI to perform key CV tasks. Subsequently, the process of training a DL model and the various techniques that can be used to optimize inference speed are explained.

2.1 Computer Vision

Computer Vision is a simulation of biological vision using computers and related equipment. Its main task is to obtain the three-dimensional information of the corresponding scene by processing collected pictures or videos [31].
CV is currently widely applied in fields such as biometry, quality analysis, smart surveillance, telemedicine, autonomous vehicles, space debris recognition and many others.

2.1.1 Computer Vision techniques

The traditional approach in CV is to use well-established mathematical algorithms and statistics to process images or videos. In particular, for object detection tasks, which will be later described, feature descriptors such as SIFT, SURF, etc., are used [44].
The *Scale Invariant Feature Transform* (SIFT) transforms an image into a large collection of local feature vectors, each of which is invariant to image translation, scaling, and rotation, and partially invariant to illumination changes and affine or 3D projection. The scale-invariant features are efficiently identified using a staged filtering approach. The first stage identifies key locations in scale space by looking for locations that are maxima or minima of a Difference of Gaussian (DoG) function. In particular, this function represents an approximation of the Laplacian of Gaussian, obtained by subtracting two Gaussian-blurred versions of the image at different scales. This approach allows for the identification of keypoints that are stable across different image scales.
From there, each relevant point is used to generate a feature vector that describes the local image region sampled relative to its scale and position. The features achieve

partial invariance to local variations, such as affine or 3D projections, by considering image gradient changes [42]. The gradient of the image is obtained by taking the partial derivatives of the image, allowing for an interpretation of the intensity changes in the image. These gradients are then used to build a histogram of gradient orientations in the local neighborhood around the keypoint.
Speeded Up Robust Features (SURF) is a CV technique that employs a simplified approximation of the Hessian matrix to emulate the operation of the DoG, leveraged by SIFT as a Laplacian-based detector.
One advantage of SURF is its computational speed. This method is, in fact, also referred to as the "Fast-Hessian" detector, enhanced by the use of integral images, which enable the rapid calculation of sums in rectangular regions by precomputing cumulative pixel values at each image position.
This allows SURF to quickly evaluate Haar-like features within a local window around each keypoint. Instead of iterating over each pixel individually, the integral image provides a way to obtain the sum of any rectangular region in constant time, making it highly efficient for real-time applications [13].
One of the most common CV algorithms used in traditional CV techniques is *Edge Detection*. The main task of edge detection is to locate and identify sharp discontinuities arising from abrupt changes of pixel intensity. These changes characterize the boundaries of objects in a scene. Edges give boundaries between different regions in the image, used to identify objects for segmentation and matching purpose. Various types of operators are available for edge detection and they can be classified into two categories. The first category involves first-order derivatives, where the input image is convolved with an adapted mask to generate a gradient image, with edges detected by thresholding. In the second category, second-order derivatives are used, based on the extraction of zero-crossing points, which indicate the presence of maxima in the image [15].

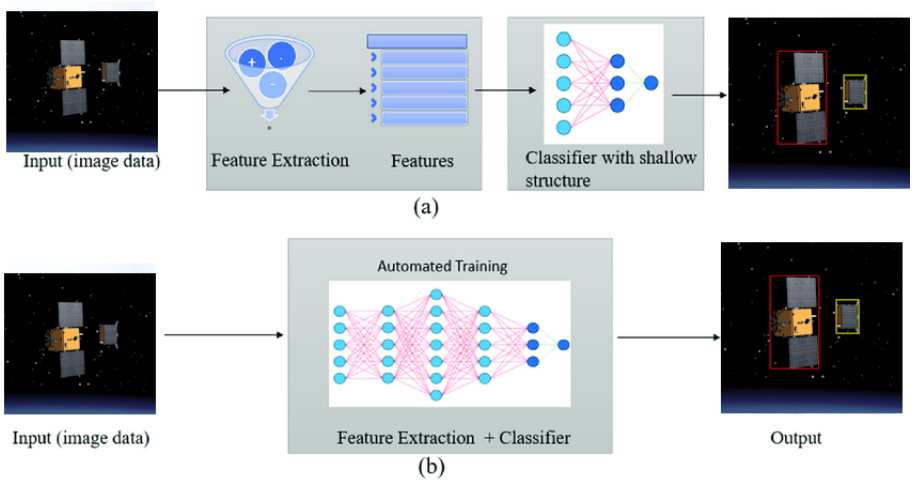


Figure 2.1: (a) Traditional CV techniques. (b) CV enhanced by DL. Base image [6]

The difficulty with the traditional approaches is that it is necessary to choose which features are relevant in each given image. In a classification task, for example, as the number of classes to classify increases, feature extraction becomes more and more

cumbersome. It is up to the CV engineer's judgment and a long trial and error process to decide which features best describe different classes of objects.
The new frontier of CV algorithms based on DL models, such as the one described in the following *Convolutional Neural Network* (CNN) section, offers greater accuracy and versatility in performing the same tasks as traditional CV techniques, at the cost of requiring significant computational resources during the training process (figure 2.1)[44].
When operating in space, the satellite has to work under extreme conditions, facing for example variable illumination, noise, and atmospheric interference condition. For this reason, it is challenging for a CV engineer to identify specific features for each condition. This makes it evident that DL-based models demonstrate superior robustness and reliability for space application, provided they are trained with sufficient and diverse datasets.

2.1.2 Computer Vision tasks

CV algorithms differ from each other depending on which features the user is interested in extracting. These features could be from an image or a video, for example. Among the most common tasks that can be performed there are:

- **Image Classification:** analysis of the content of an image and associating it with a label, based on predefined categories or metrics. For example given a dataset of images related to space operations, the algorithm is able to assign a category to each one of the images, such as "satellite fragment", "rocket component", "astronaut" and so on.

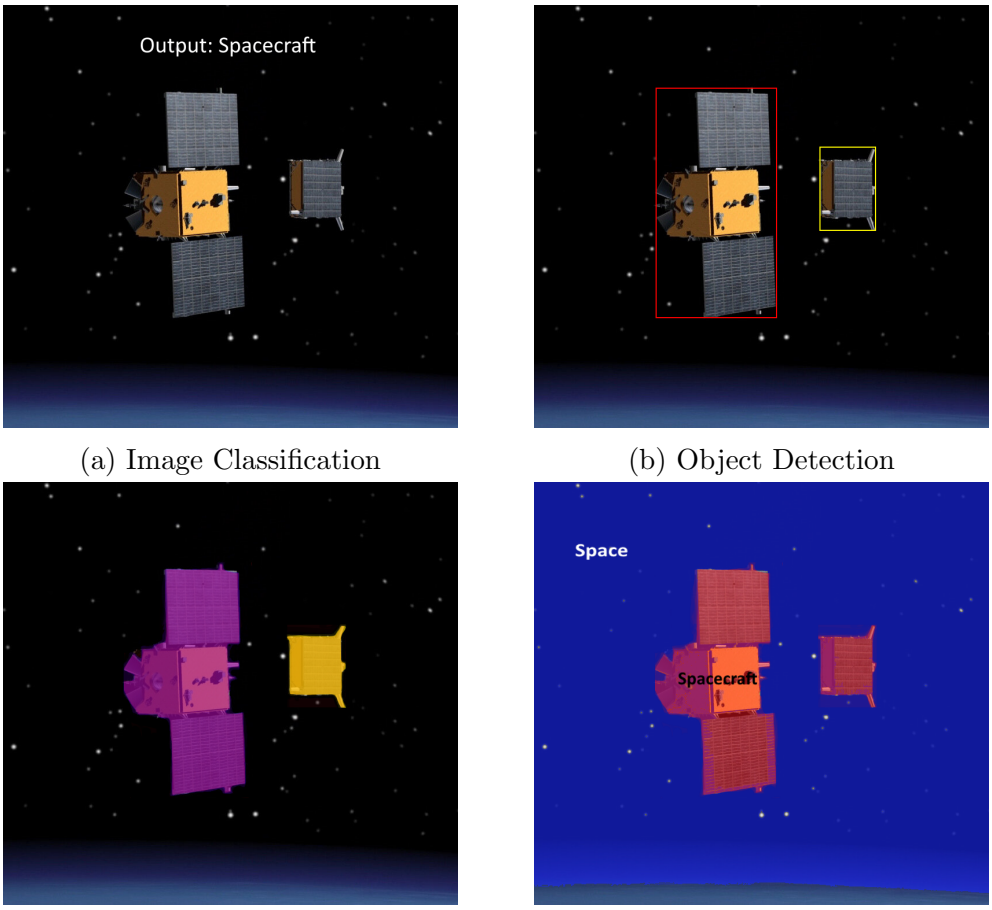
- **Object Detection:** identification of one or more entity in a picture. It combines image classification with object localization, generating rectangular regions, called "bounding boxes", in which objects are located.

- **Image Segmentation:** partition an image into multiple parts or regions, often based on the characteristics of the pixels in the image. Conventional image segmentation algorithms process high-level visual features of each pixel, like color or brightness, to identify object boundaries and background regions.
  The image Segmentation can be further divided in semantic or instance segmentation [66]. The main difference between the two is that semantic segmentation associates a semantic class to every pixel grouping all objects of the same class together, while instance segmentation identifies and delineates individual instances of objects within an image even if they belong to the same class.

- **Face Recognition:** recognition of the characteristic features of people's faces.

- **Action Recognition:** identification of one or more entities and their relationship in time and space, in order to identify and describe specific actions.

- **Visual Relationship Detection**: comprehension of the relationship between the objects of an image.



(a) Image Classification

(b) Object Detection

(c) Instance Segmentation

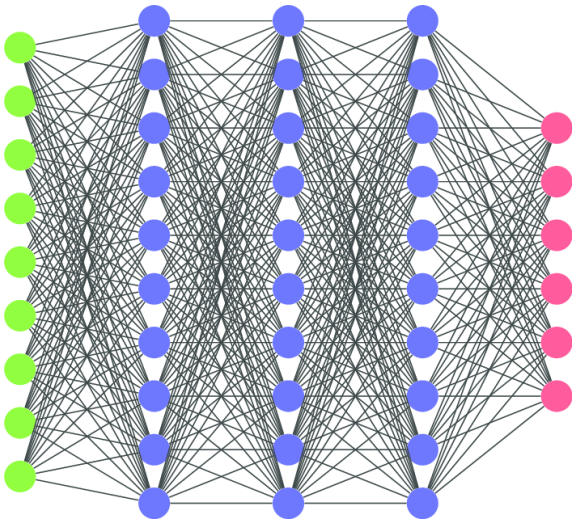
(d) Semantic Segmentation

Figure 2.2: Examples of different tasks performed by computer vision algorithms (base image [39])

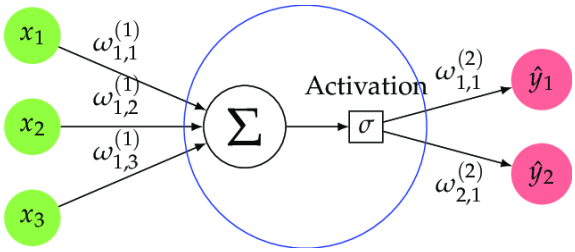
2.2 Deep Learning

2.2.1 Neural Networks

As previously mentioned, neural networks (NNs) form the backbone of DL models, and in order to develop a vision-based detector, it is essential to understand how these NNs function. Figure 2.3a shows a typical NN architecture. Here the green layer represents the input layer, the blue are the hidden layers and finally the pink one the output layer.
The neurons, represented by blue circles, are the building blocks of a network. Each neuron sums its input signals weighted by the connection strengths (the weights) and applies an activation function σ to generate an output signal, as shown in figure 2.3b. There are different types of activation functions, going from Linear to Nonlinear ones. The most commonly used nonlinear activation functions are the sigmoid, tanh, or ReLU (Rectified Linear Unit) [56]. Those are typically applied in both hidden and output layers, whereas the linear activation function is mostly



(a) Typical architecture of a Neural Network



(b) Neuron action

Figure 2.3: Graphical interpretation of the action of a Neural Network [50]

used in the output nodes. The nonlinear activation functions allow the model to generalize and adapt to a variety of data, enabling complex modeling.
One pass of the entire dataset through the NN, including both forward and backward passes, is known as an epoch. Multiple epochs are used to pass the dataset multiple times.
To ease computational pressure, the epoch can be split into batches which dictate the number of iterations needed to complete one epoch. For example, 2000 examples that are split into 500 batches would require four iterations to complete one epoch. However, using too many small batches can increase the computational overhead and lead to less stable training.
The term "deep" refers to the number of layers in the network. In particular, the greater the number of layers, the deeper the network. While traditional NN generally contain only 2 or 3 layers, deep networks can have hundreds of them, allowing for a detailed and complex modeling.

2.2.2 Convolutional Neural Networks

A CNN is a specialized NN that has been developed to work with data structured in grids or matrices such as images, audio and signals. As a traditional NN, its

structure is composed by an input, an output and several hidden layers, that have to perform the operation of convolution, pooling and fully connected transformations. When talking about CNNs, it is common to refer to a group of layers that perform the same specific action as a *block* [9]. The main task of the convolutional

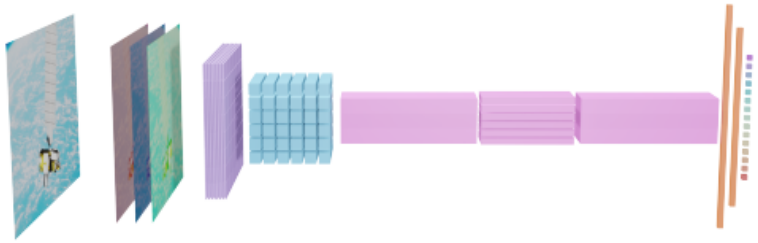


Figure 2.4: Typical architecture of a Convolutional Neural Network [30]

layer is to extract significant features from the input. For example, a color image can be represented as a three dimensional matrix of pixels, where each dimension corresponds to the height, width, and depth (the three channels: red, green, and blue). Now introducing the filter, or kernel, of the convolutional layer as a smaller matrix, with respect to the input matrix, the convolutional operation is obtained by sliding the filter over the entire image.
In particular during the convolution process, the filter is applied to a small region of the image (the receptive field) and computes a dot product between the pixel values in the image and the values in the filter. The result of this operation is placed in a feature map. Afterward, the filter shifts by a fixed amount, called the stride, and the process repeats until the entire image is covered. Each point in the feature map represents a feature extracted from the corresponding portion of the image. After each convolution operation, a CNN applies a ReLU transformation to the feature map, introducing nonlinearity to the model.
The pooling layers, also known as down-sampling layers, conduct dimensionality reduction, reducing the number of parameters in the input. Similar to the convolutional layer, the pooling operation sweeps a filter across the entire input, but the difference is that this filter does not have any weights. Instead, the kernel applies an aggregation function (such as maximum or average) to the values within the receptive field, populating the output array [32].
The fully connected layer, or dense layer, has the task to capture global patterns and relationships in the input data by connecting every neuron from the previous layer to every neuron in the fully connected layer. The primary function of the fully connected layer is to perform high-level reasoning and decision-making based on the features extracted by the preceding layers. It accomplishes this by learning complex non-linear mappings between the input and output data. Each neuron in the fully connected layer receives inputs from all the neurons in the previous layer and produces an output by applying a set of weights and biases, followed by an activation function, following then procedure of a conventional NN. An example of a typical architecture are shown in figure 2.4.
When building a CNN, or more generally a NN, there are many variables to keep

in consideration, such as how many groups, layers and neurons use, without considering the initialization of the weights and the selection of the proper activation functions. One factor that has contributed to the widespread adoption of computer vision is the availability of pre-trained models developed by large companies. These models are freely available and adjustable for personal tasks, making this technology more approachable.

EfficientNet

The state of the art in image classification is the family of algorithms called EfficientNet, which achieves 84.3% top-1 accuracy on the ImageNet dataset (1.2 million images classified into a thousand categories)[22] , while being 8.4 times smaller and 6.1 times faster in inference than the best existing CNN [58]. It is common practice to develop CNNs at a fixed resource budget, and then, scale them up for better accuracy if more resources are available. However, achieving an optimal balance between network width, depth, and resolution during scaling is challenging. In the EfficientNet family, such balance is achieved by simply scaling each network dimension with a constant ratio. This procedure is called *compound scaling* method and is well described in [58].

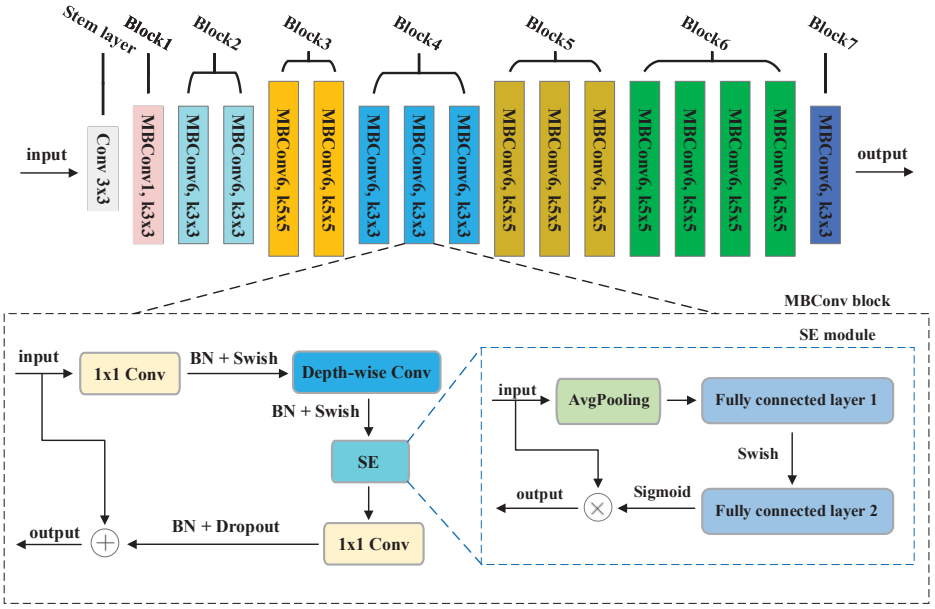


Figure 2.5: EfficientNet-B0 architecture [71]

The main building block of the EfficientNet architecture is the Mobile Inverted Bottleneck Convolution (MBConv), based on the concept introduced in MobileNet [52]. The MBConv block, as shown in figure 2.5, consists of three convolutional layers followed by a squeeze-and-excitation (SE) block.
At first, a point-wise convolution is performed that expands the number of channels, allowing for more complex feature interactions. By increasing the dimensionality of the feature map, the network is able to learn a richer representation.

The expansion factor of each layer is specified in the name of the group by the number following MBConv. For example, MBConv6 means an expansion factor of 6.
Next, a depth-wise convolution is applied, where instead of processing all channels simultaneously, each channel is convolved separately.
After each convolutional operation, Batch Normalization is applied to stabilize training, followed by the Swish activation to introduce non-linearity, whose behavior is described in equation 2.1:

$$f_{Swish} = \frac{1}{1 + e^{-\beta x}} \tag{2.1}$$

where β is a parameter that can be learned during the CNN training.
Finally, the SE block tells the model which features are essential and suppresses the less relevant ones. It uses global average pooling to reduce the spatial dimensions of the feature map to a single channel, followed by two fully connected layers to learn channel-wise weights, effectively recalibrating the importance of each channel.
The EfficientNet-B0 architecture is designed to be scalable, meaning that its depth, width, and resolution can be uniformly scaled up to create larger, more accurate models (like EfficientNet-B1, B2, etc.) while maintaining efficiency.
In this thesis project, to develop an object detection model, the pre-trained EfficientNet-B0 model, whose architecture is shown in figure 2.5, was used, since it provides a good balance between computational resources and accuracy.

You Only Look Once

In the field of object detection, the *You Only Look Once* (YOLO) family of algorithms stands out as one of the most performant. Despite being single-stage detectors, their accuracy is often comparable to that of two-stage detectors. For instance, YOLO achieves an accuracy of 63.4% compared to the 70% of Fast R-CNN, yet it operates nearly 300 times faster in inference [27]. This makes YOLO models the state of the art for real-time applications. Unlike traditional approaches, YOLO formulates object detection as a regression problem rather than a classification task, using a convolutional neural network to predict bounding boxes and class probabilities in a single step.
Given its efficiency and accuracy, YOLO is used in this thesis as a benchmark to evaluate the developed model. In particular, the YOLOv8n model from Ultralytics [62] has been selected for evaluation, where 'n' stands for nano, indicating the smallest and fastest variant among the YOLOv8 models.
As shown in figure 2.6, the YOLOv8 architecture is divided into three main components: backbone, neck, and head. The backbone, possibly an advanced version of CSPDarknet or another efficient architecture [59], is responsible for extracting relevant features from the input image through convolutional layers. It incorporates Convolution, Batch Normalization, and SiLU activation function (CBS), which enhances stability and convergence. The backbone also includes C2f, a CSP Bottleneck with Fusion, which reduces redundant computations by splitting and merging feature pathways. Additionally, the Spatial Pyramid Pooling - Fast Compact (SPPFC) module expands the receptive field and preserves spatial information, improving the model's ability to recognize objects at different scales.

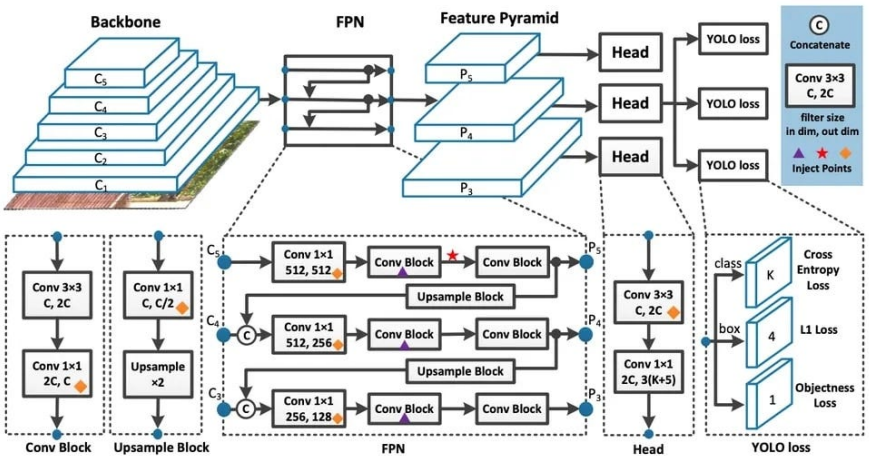


Figure 2.6: YOLOv8 architecture [7]

The neck serves as a bridge between the backbone and the head, refining the extracted features before final predictions. It integrates structures such as the Feature Pyramid Network (FPN), which enhances multi-scale detection by combining low-level detailed features with high-level abstract representations, and the Path Aggregation Network (PAN), which enables bidirectional information flow. This allows finer details from lower layers to influence higher-level feature maps, improving both localization accuracy and bounding box quality.
Finally, the head is responsible for generating the final detection. YOLOv8 supports both anchor-based and anchor-free methods, providing greater flexibility in balancing speed and accuracy.

2.3 Neural Networks learning theory

In order to work properly, a NN must undergo a training process, which results in the optimization of parameters such as weights and biases, so that it is ultimately able to learn and solve a specific task [11]. During training, ML algorithms process large amounts of historical data to identify patterns through inference.
Depending on the data provided and the task that the NNs have to perform, it is possible to divide the learning process into supervised and unsupervised learning [57].
Unsupervised learning refers to training with unlabeled data. In particular, the model examines new data and establishes meaningful relationships between the unknown input and its underlying structure. These models rely solely on input data to identify patterns and structures within it, and they are widely used for knowledge extraction, data compression/denoising, grouping, or clustering tasks. In general, models that undergo unsupervised learning can be predictive, if they make predictions on new data; descriptive, if they aim to understand and represent the data; or both. The evaluation of their performance depends on whether the goal is to reproduce existing knowledge or to acquire new insights.
Supervised learning, on the other hand, relies on labeled data, where both the input

and the corresponding output are provided to the algorithm during training. Given a sufficient amount of labeled data, a supervised learning system can, over time, recognize patterns and structures. This type of learning is mainly used for classification or regression tasks, depending on whether the output consists of a discrete set of values, the classes, or a continuous variable, respectively.

2.3.1 Supervised Learning

To truly understand the principles behind the NN training, the logistic regression model, characterized by a relatively low level of complexity, is introduced. This learning algorithm is used when the output labels $\hat{y}$ in a supervised learning problem are binary, taking values of either zero or one. So the goal of training this model is that, given an input vector x, the output $\hat{y}$, should be the most accurate possible probability that determines at which class the x belongs, specified by the correct label y.
During a *forward pass* all the n input pass through the algorithm ending with n evaluation of probabilities $\hat{y}$ and metrics. Considering now the parameters of the logistic regression, W (weights), b (bias) and activation function σ, here a sigmoid, the output of the model is described by equation 2.2.

$$\begin{aligned} z &= W^T x + b \\ \hat{y} &= \sigma(z) \end{aligned} \tag{2.2}$$

A metric to evaluate how well the process is doing within each training example is the loss function. In logistic regression, a logarithmic function, described by equation 2.3, is usually used since it is guaranteed to be convex for all input values, containing only one minimum, allowing to run the gradient descent algorithm, as it will be discussed later.

$$\mathcal{L}(\hat{y}, y) = -(y\hat{log(y)} + (1-y)log(1-\hat{y})) \tag{2.3}$$

It is now time to introduce the cost function $J(W, b)$, that is an appropriate parameter to estimate the quality of the whole forward pass. The cost function is indeed the mean of all the loss values computed for the n probabilities, here defined in equation 2.4.

$$J(W, b) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \tag{2.4}$$

To summarize, the loss function can be interpreted as a measure of how accurate are the predictions on a single training example, while the cost function does the same but on the entire training set.
After completing the *forward pass*, the next step in the training process is the *backward pass*. Once J is defined, it is time to minimize it by adjusting the parameters W and b, on which it depends. To do so, the iterative optimization algorithm *Gradient Descent* is applied.
The common procedure is to initialize the whole process by selecting the parameters equal to zero or picking them randomly. Once W and b are defined, the J is evaluated at these values. Then, from this location, the algorithm takes a step in

the steepest downhill direction of the cost function, i.e., toward a smaller value of J. The size of how big is the step taken set by the learning ratio α, which is used to define the update rule for the parameters described in equation 2.5.

$$\begin{aligned} W_i &= W_i - \alpha dW_i \\ b &= b - \alpha db \end{aligned} \tag{2.5}$$

In particular, in equation 2.5 the subscript i stands for the generic element of the weights and dW and db are obtained with equation 2.6.

$$\begin{aligned} dW &= \frac{\partial J(W,b)}{\partial W} \\ db &= dz = \hat{y} - y \end{aligned} \tag{2.6}$$

Once the parameters are updated, a new forward pass is performed reiterating the same process until the minimum of J is reached. A visual representation of what is described above is shown in figure 2.7.

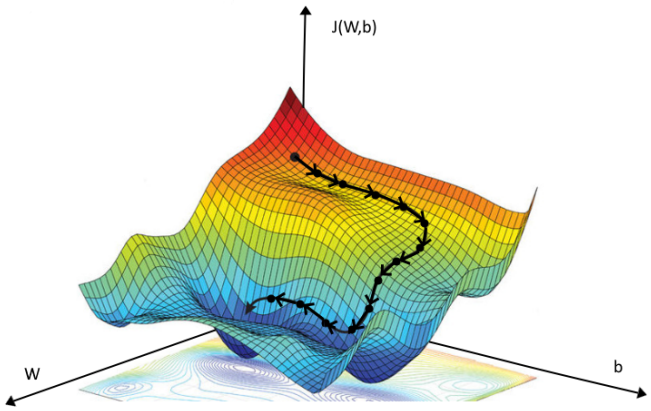


Figure 2.7: Visual representation of gradient descent applied to a function with two parameters to be optimized. This is just an example, as gradient descent can also be applied to functions with multiple parameters (base image[10])

This process can now be generalized to a DNN with L layers, and $n^{[l]}$, $a^{[l]}$, $g^{[l]}$, and $W^{[l]}$ representing the number of units, the activations, the activation functions, and the weights of the l-th layer, respectively.
Considering just a training example, it is possible to describe the forward pass through a generic layer by applying equation 2.7.

$$\begin{aligned} z^{[l]} &= W^{[l]} a^{[l-1]} + b^{[l]} \\ a^{[l]} &= g^{[l]}(z^{[l]}) \end{aligned} \tag{2.7}$$

Meanwhile, for the backward propagation, equation 2.8 hold true, always applied to

a specific l layer, where $*$ represent the element wise product.

$$\begin{aligned} dz^{[l]} &= da^{[l]} * g^{[l]}(z^{[l]}) \\ dW^{[l]} &= dz^{[l]} a^{[l-1]T} \\ db^{[l]} &= dz^{[l]} \\ da^{[l-1]} &= W^{[l]T} dz^{[l]} \end{aligned} \tag{2.8}$$

The process of forward and backward propagation is applied iteratively across all layers in the network. First, in the forward pass, from left to right, starting from the input layer, the activations $a^{[l]}$ are computed layer by layer until the final output layer, where the cost function is evaluated. Second, from right to left, the backward pass propagates the error $dz^{[l]}$ through each layer to compute gradients to update parameters $W^{[l]}$ and $b^{[l]}$ [11].
By combining the computations across all layers, the network forms a pipeline where each layer's output serves as the input to the next. This interconnected structure allows the network to map input data to predictions during the forward pass. In the backward pass, errors are distributed across the layers, enabling the parameters to be iteratively updated to minimize the cost function.
Finally the entire process can be vectorized, allowing the training for the entire dataset.

2.3.2 Loss Functions

A typical loss function used in CNNs for tasks involving continuous output predictions, such as object detection based on keypoints, is the Mean Squared Error (MSE). It creates a criterion, described in equation 2.9, that measures the mean squared error (squared L2 norm) between each element in the input $\hat{y}$ and target y.

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \tag{2.9}$$

From the MSE, it is straightforward to derive the Root Mean Squared Error (RMSE), which evaluates the square root of the average squared errors (equation 2.10).

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2} \tag{2.10}$$

The Binary Cross Entropy (BCE) is a performance measure for classification models that output predictions with probability values typically between 0 and 1. This prediction value corresponds to the likelihood of a data sample belonging to a particular class or category. The BCE $\ell(x, y)$ can be described by equation 2.11 [49].

$$\begin{aligned} l_n &= -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right] \\ \ell(x, y) &= L = \left[l_1, \ldots, l_N \right]^\top \end{aligned} \tag{2.11}$$

Subsequently, this loss function can be reduced by either summing or averaging the elements of $\ell(x, y)$, where x and y represent the prediction and the target, respectively, and w represents the weights associated with each class. The BCE can be combined with the sigmoid activation function σ, as described in equation 2.12. This combination is used when the model outputs logits, i.e., a continuous value rather than a probability. In this case, the probability is computed precisely through the sigmoid function, represented in equation 2.13 and the loss function l_n is referred to as BCE with logits.

$$\begin{aligned} l_n &= -w_n \left[y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n)) \right] \\ \ell(x, y) &= \left[l_1, \ldots, l_N \right]^\top \end{aligned} \tag{2.12}$$

$$\sigma(x_n) = \frac{1}{1 + e^{-x_n}} \tag{2.13}$$

The Focal Loss (FL) is one of the most common loss functions when the algorithm needs to learn dense classification scores. It is particularly suited to address one-stage object detection scenario, where an extreme imbalance between foreground and background classes in the training data exists, specifically when the background class dominates [40]. FL is evaluated thanks to equation 2.14:

$$\mathrm{FL}(p) = -(1 - p_t)^\gamma \log(p_t), \quad p_t = \begin{cases} p, & \text{when } y = 1 \\ 1 - p, & \text{when } y = 0 \end{cases} \tag{2.14}$$

where $y \in \{0, 1\}$ specifies the ground-truth class, $p \in [0, 1]$ denotes the estimated probability for the class with label $y = 1$ and γ is the tunable focusing parameter. From equation 2.14, it is possible to notice that the FL intensifies the loss for data points that have a large difference between the predicted and actual outputs, effectively making the NN focus more on hard-to-classify examples.
The Distribution Focal Loss (DFL) is a variation of FL used specifically for bounding box (BB) regression [40]. Where the term BB refers to the box with the smallest size (of area, volume, or hypervolume in larger dimensions) within which all relevant points of the object to be detected are contained.
DFL improves localization accuracy by enhancing the gradient condition, leading to better performance in handling difficult-to-classify data points. It further improves on FL by modeling the locations of bounding boxes as general distributions, while forcing the networks to rapidly focus on learning the probabilities of values close to the target coordinates. Instead of a single predicted value, DFL ensures the model learns a distribution over possible values.
Conventional operations of BB regression model the predicted labels with a Dirac delta distribution δ described in equation 2.15.

$$\int_{-\infty}^{+\infty} \delta(x - y) dx = 1 \tag{2.15}$$

It is possible then to define the regressed label y as in equation 2.16.

$$y = \int_{-\infty}^{+\infty} \delta(x - y) x \, dx \tag{2.16}$$

However, this approach relies on priors like Dirac delta or Gaussian distributions. Another possible way of describing the predicted label is through the general distribution $P(x)$. The estimated regression value $\hat{y}$ is computed by integrating over the range of y, from y_0 to y_n, which represent the minimum and maximum values of the regressed label, as shown in equation 2.17.

$$\hat{y} = \int_{y_0}^{y_n} P(x)x\,dx \tag{2.17}$$

$P(x)$ is generally implemented through a softmax layer $S(\cdot)$, where the values $P(y_i)$ are denoted as S_i.
However, this approach introduces some limitations. In fact, there exist infinite combinations of values for $P(x)$ that can produce the same result for $\hat{y}$, leading to inefficiencies in learning.
The DFL is a robust solution to this problem. By emphasizing the learning of values around the target y, it increases the probabilities of values close to y. Specifically, it focuses on the two nearest values, y_i and y_{i+1}, and encouraging higher probabilities for them. This operation is described in equation 2.18.

$$\text{DFL}(S_i, S_{i+1}) = -\left((y_{i+1} - y)\log(S_i) + (y - y_i)\log(S_{i+1})\right) \tag{2.18}$$

The global minimum solution of DFL guarantees that the estimated regression target $\hat{y}$ is infinitely close to the corresponding label y.
Finally the Complete Intersection over Union (CIoU) loss function is another widespread loss function used to help convergence in the object detection models [63]. It is based on the Intersection over Union (IoU) metric that will be described further in this chapter.
The CIoU loss function, defined in equation 2.19, takes into account not only the intersection of the areas of the predicted and ground truth BB but also the Euclidean distance d between their centers and the minimum diagonal distance C of the rectangle that encloses both BBs.

$$L_{CIoU} = 1 - IoU + \frac{d^2}{C^2} + \alpha v \tag{2.19}$$

In particular v describes the aspect ratio difference between the two BBs, and α is a trade-off parameter, function of IoU, as defined respectively in equations 2.20 and 2.21.

$$v = \frac{4}{\pi^2}(arctan\frac{w^{gt}}{h^{gt}} - arctan\frac{w}{h})^2 \tag{2.20}$$

$$\alpha = \frac{v}{(1 - IoU) + v} \tag{2.21}$$

Thanks to the definition of the CIoU loss given in equation 2.19, it is possible to observe that the aspect ratio factor is less significant in the case of no overlap and more important in the case of a greater overlap.

2.3.3 Optimization algorithm

Depending on the situation, there are several techniques to perform Gradient Descent, ranging from the most general one, described in equation 2.6, to more advanced strategies such as the Adaptive Moment Estimation (Adam) and Stochastic Gradient Descent with momentum (SGDM) optimization algorithms [69].
ADAM is an adaptive learning rate algorithm designed to improve training speed and achieve faster convergence. In the standard gradient descent algorithm, the learning rate α is fixed. Typically, α starts with a higher value, and then it is manually adjusted in steps or according to a learning schedule. A lower learning rate at the onset would lead to very slow convergence, while a very high rate at the start might miss the minimum.
The Adam optimizer instead is able to adapt the learning rate for each parameter that needs to be optimized [26]. To do so, momentum is introduced to speed up the training by accelerating gradients in the right directions, adding a fraction of the previous gradient to the current one. For example, let's say a gradient has been consistently pointing in the same direction. The momentum term, proportional to the previous gradients, will accumulate and accelerate the optimization in that direction. So, being θ the model parameters, the gradient of the loss function at the t-th iteration is obtained using equation 2.22.

$$g_t = \nabla_\theta \mathcal{L}(\theta_t) \tag{2.22}$$

The moments, computed by ADAM, are the first order m_t and second order v_t momentum, which represent respectively the mean of gradients and the the uncentered variance of gradient, are defined by equation 2.23.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \end{aligned} \tag{2.23}$$

It is now necessary to introduce the correction to the moments, as described in equation 2.24, due to the fact that they have been initialized at zero. As a result, their values may differ significantly from the true ones, being biased towards zero.

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{2.24}$$

Finally, the last step is the update of the model parameters, as shown in equation 2.25.

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.25}$$

Where β_1 and β_2 are hyper-parameters that control the exponential decay of the moments and ϵ is term of numerical stability.
Considering now the definition of the gradient of the cost function g_t given in equation 2.22, the update of the parameters performed by SGD at each t step is described by equation 2.26.

$$\theta_{t+1} = \theta_t - \alpha g_t \tag{2.26}$$

A variation of SGD is the SGDM, where the momentum is introduced to speed up learning and avoid wide oscillations during the training process. The concept behind the SGDM algorithm is that, during the update, not only the gradient of the current step is used, but also a moving average of the gradient from the previous steps. It is then possible to define the momentum m_t as shown in equation 2.24, from which the update rule, described in equation 2.27, is derived.

$$\theta_{t+1} = \theta_t - \alpha m_t \tag{2.27}$$

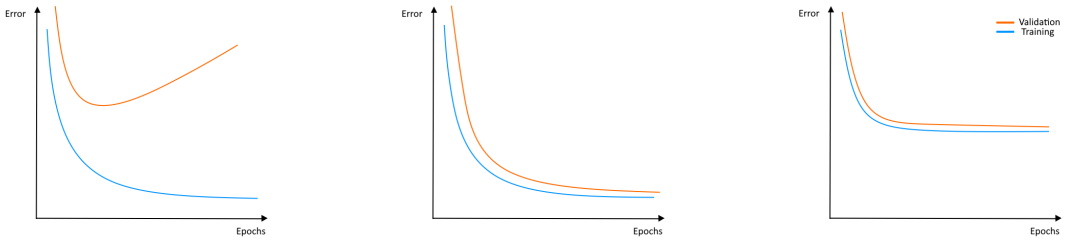
In conclusion, SGDM can achieve more accurate minima in the loss function at the cost of longer training. This means that ADAM converges much faster to a local minimum, often oscillating around it, at the cost of lower precision. Furthermore, the choice of hyperparameters in ADAM is less critical, even though their number is higher, compared to SGD and SGDM, where the choice of α is crucial.

2.3.4 Training limitations

A training session is defined as successful if the model correctly fits the test inputs. If this is not the case, the model may suffer from underfitting or overfitting problems.
A model experiences overfitting if it is unable to generalize well from the training data to unseen data. As a result, the model performs consummately on the training set while fitting ineffectively on the testing set. This happens because an over-fitted model experiences issues adapting to bits of the data in the testing set, which might be unique about those in the training set.
This phenomenon may be correlated with the training dataset used. A dataset that is too small or noisy may lead the model to learn the noise component as well, causing it to look for it in the test data. Another factor that can lead to overfitting is the model's hypercomplexity.
There are different techniques to prevent overfitting problems, such as early stopping, network reduction, dataset expansion, and regularization. During regularization, a penalty is added to the complexity of the model, reducing the magnitude of the weights [48].
Underfitting, on the other hand, as its name suggests, is the opposite problem. It describes a model that does not capture the underlying relationship in the dataset on which it is trained. As a result, it will neither perform well on the training set nor generalize well to unseen data
Some solutions include increasing the model complexity, reducing regularization, and adding features to the training data.
In figure 2.8, the typical trends of training and validation losses are shown for overfitted models 2.8a and underfitted models 2.8c, respectively.

2.3.5 Accuracy metrics

To assess the performance of a DL model, it is necessary to define some measurable and comparable quantities called metrics.
The Frames Per Second FPS metric is of particular importance in all those systems



(a) Overfitting (b) Correct fitting (c) Underfitting

Figure 2.8: Examples of Overfitting and Underfitting

that have to operate in real-time. It is a measure of the model's speed and efficiency, indicating how quickly it can handle incoming images and generate object detection results.
The most common object detection metric is the *Intersection over Union* (IoU). Exploiting the bounding box coordinates, the IoU evaluates the ratio of the area of intersection to the area of the union between the predicted and ground truth bounding boxes (figure 2.9).

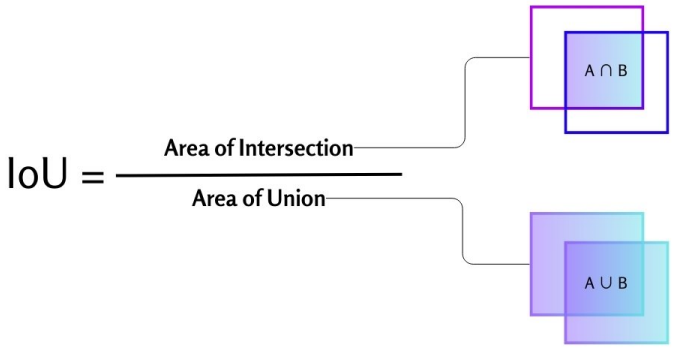


Figure 2.9: Visualization of the IoU formula [16]

Another common measure of accuracy is the *Average Precision* (AP) [45]. To provide the correct definition of AP, it is necessary to introduce the following concepts. In the context of object detection, a true positive (TP) case occurs when there is a correct detection of a ground-truth BB. A false positive (FP) is an incorrect detection of a nonexistent object or a misplaced detection of an existing object. Finally, a false negative (FN) is an undetected ground-truth BB.
By comparing the IoU with a given threshold t, it is possible to asses if a detection is correct, $IoU \geq t$ or not, $IoU < t$. The precision (P) and recall (R) can be defined as in equation 2.28.

$$\begin{aligned} P &= \frac{TP}{TP + FP} \\ R &= \frac{TP}{TP + FN} \end{aligned} \tag{2.28}$$

Precision is the ability of a model to identify only relevant objects, representing the percentage of correct positive predictions. Recall, on the other hand, is the ability of a model to find all relevant cases, i.e., all the ground-truth BB. R represents the percentage of correct positive predictions among all given ground truths. The AP is then the area under the precision-recall curve, as defined in equation 2.29.

$$AP = \int_{r=0}^{1} p(r)dr \tag{2.29}$$

Finally, equation 2.30 describes the mean average precision (mAP), which measure the accuracy of object detectors over all classes in a specific database.

$$mAP = \frac{1}{k} \sum_{i}^{k} AP_i \tag{2.30}$$

In particular, mAP@50 is the mean of AP values calculated with an IoU threshold t of 0.5 and mAP@50-95 the mean of AP values computed over IoU thresholds ranging from 0.50 to 0.95 in steps of 0.05, as defined in equation 2.31:

$$mAP@50-95 = \frac{1}{N} \sum_{t=0.50}^{0.95} AP_t \tag{2.31}$$

where AP_t is the Average Precision computed at IoU threshold t.

2.4 Inference speed optimization techniques

The inference speed in DL refers to the time it takes for a trained model to make predictions, such as regressions or classifications, on new data.
Reducing inference time is a critical requirement especially for real-time applications. This task becomes even more important in space applications, such as image recognition for debris avoidance or collection, where onboard power is limited. Faster inference, in fact, reduces the energy consumption required for computation.
The following sections present various techniques and solutions designed to reduce the inference time of a CNN.

2.4.1 GPU acceleration

A first solution to improve inference speed is acceleration through a graphics processing unit (GPU). Unlike the central processing unit (CPU), which acts as the "brain" of a computer, optimized for executing complex instructions sequentially while managing general system operations, the GPU is specialized for parallel computations.
In general, while the CPU handles more high-level tasks and control operations, the GPU excels at specific tasks that require high computational power. This is due to a higher number of cores, which, even though simpler than CPU cores, allow for massive parallelism. In addition, the GPU has access to dedicated memory (VRAM),

which enables faster computation. These cores work together by distributing processing tasks across many units simultaneously, significantly improving performance [18].
In the context of ML, where each inference involves a large number of parameters, GPU acceleration can significantly reduce inference time. Technologies such as Compute Unified Device Architecture (CUDA), CUDA Deep Neural Network library (cuDNN) and Tensor Runtime (TensorRT) further optimize inference performance by exploiting the GPU architecture.
However, this computational speed comes at the cost of high power consumption. Additionally, for very small models, the data transfer latency between the main RAM and VRAM can sometimes negate the speed benefits.
This technology is, of course, also leveraged during the training process, where even clusters of GPUs are used to process batches of data simultaneously, further speeding up the entire process.

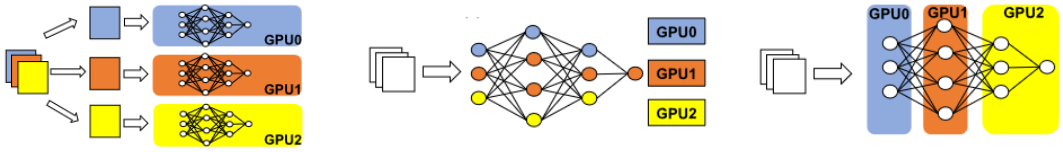
2.4.2 Parallelization

The parallelization technique exploits multiple processing units such as GPUs or multi-core CPU to distribute the computational workload. Through out this process a lower inference time is achieved, guaranteeing a reduced latency and the ability to handle models that could exceed the memory of a single device [68]. The most common methods for performing inference parallelism are as follow:

- **Data Parallelism:** during data parallelism the input data is split into mini-batches which are distributed across different devices working in parallel (figure 2.10a). When the output is computed it is necessary to perform an aggregation of the results. This method mainly benefits the training process, where large amounts of data are required. Here the synchronization is performed at the end of each batch processing to gather the gradients and update the model weights. It can be useful during inference if more batches of input data are involved, this does not strictly imply a faster inference but allows for a throughput improvement.

- **Model Parallelism**: The model parallelism method consists in splitting the model and allocating different parts of it across multiple GPUs. During this process, the entire input dataset is simultaneously stored on each device.
  This partitioning can be done in two ways. The first consists in horizontally partitioning the model, meaning that neurons from different layer are distributed across multiple GPUs. This approach is illustrated in figure 2.10b. The second method divides the model layer-wise, as shown in figure 2.10c, where entire layers are assigned to individual GPUs, and different GPUs handle different layers.

- **Pipeline Parallelism**: In pipeline parallelism, different stages of the process are carried out in different devices, but concurrently. For example, different layers of the ML model can be placed in different devices, forming a pipeline [43].



(a) Data parallelism (b) Tensor parallelism (c) Model layer-wise division

Figure 2.10: Principal inference parallelization methods[68]

2.4.3 Pruning

Model pruning is a technique used to remove unimportant parameters from NNs, enhancing efficiency without significantly compromising performance. It balances model accuracy with size reduction, making it ideal for deployment in constrained environments or real-time applications.
NN pruning has proven to be an effective method for reducing memory usage and computational time during inference while maintaining comparable, or even superior performance, to the original DNNs. Pruning can be applied both during training and in the post-training phase, depending on the available computational power and the complexity of the operations [20].
There are two main methods to operate pruning on a NN: unstructured and structured pruning.
Unstructured pruning involves zeroing out individual weights in the weight matrix if they fall below a defined threshold, which can be determined based on the magnitude or gradient. Since all calculations are performed before pruning, this method provides minimal latency improvement. Essentially, it removes individual connections within the network.
Structured pruning, on the other hand, removes entire structured groups of weights, such as filters, kernels, or channels. This approach significantly reduces the number of computations required during the forward pass, leading to improved inference speed.
A graphical explanation of the different operations carried out with these two methods is given in figure 2.11.

2.4.4 Quantization

Quantization is a procedure that reduces the precision used to represent NN parameters, usually from n bits to m bits, where $n > m$ [65].
The standard format for training CNNs is usually 32-bit floating-point representation (fp32). For this reason, one of the most common quantization processes is the conversion of CNNs to a 16-bit floating-point (fp16) or an 8-bit integer (int8) representation.
The beneficial effect of quantization in speeding up inferences derives from the fact that integer arithmetic is less complex than floating-point arithmetic. With fewer bits to compute and an efficient numerical representation, executing an 8-bit integer NN significantly lowers latency, energy consumption, and resource utilization, albeit at the cost of lower inference accuracy.

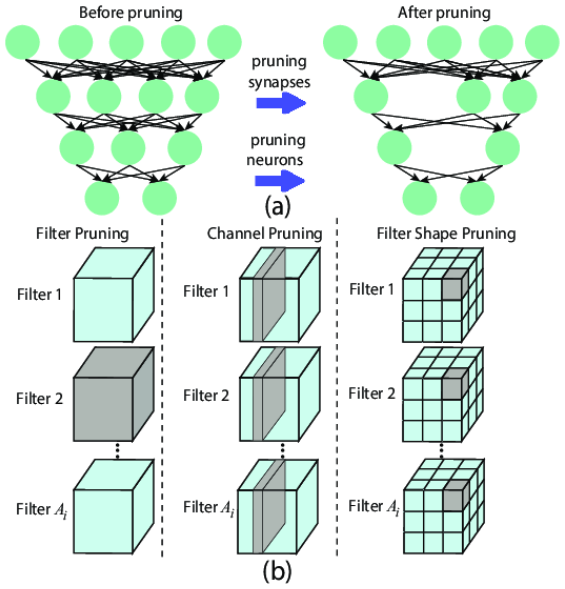


Figure 2.11: Pruning methods. a) Unstructured pruning, b) Structured pruning[64]

The mapping from real numbers r to quantized integers q is described in equation 2.32 [21].

$$\begin{aligned} q &= round\left(\frac{r}{S} + Z\right) \\ r &= S(q - Z) \end{aligned} \tag{2.32}$$

where S and Z are scale and zero point quantization parameters. In the case of floating-point distributions, S and Z can be calculated based on the distribution of the floating-point values, as shown in equation 2.33.

$$\begin{aligned} S &= \frac{r_{max} - r_{min}}{q_{max} - q_{min}} \\ Z &= round\left(q_{max} - \frac{r_{max}}{S}\right) \end{aligned} \tag{2.33}$$

If the quantization scheme is symmetric, the quantization is centered around zero, which is achieved by setting Z to zero. On the other hand, in the case of asymmetric quantization, the scale and shift values are adjusted to better match the distribution of the data. This approach allows for a more efficient use of the available bit-width and helps reduce quantization error, though it may come with an increase in computational complexity.
Model quantization techniques can be broadly classified in two categories.
The *Post-Training Quantization* (PTQ) is a quantization operation performed on a pre-trained floating-point model. Before the quantization process begins, a layer fusion operation is carried out for convolution, batch normalization, and activation functions to reduce the number of potential sources of quantization errors.
Unlike weights, quantizing activations can be challenging because their distribution is unknown. For this reason, a calibration dataset is introduced to collect the dynamic ranges of the weights and biases in the convolutional and fully connected layers of the network, as well as the dynamic ranges of the activations in all layers

of the network. The selected data can be unlabeled, provided it is representative of the actual distribution expected during inference operations. There is no fixed recommended size for the calibration dataset; the choice depends on the specific case. Research [21] shows that a calibration dataset consisting of a few hundred to a thousand shuffled images can generalize the classification accuracy of a network trained on the ImageNet dataset, which contains approximately one million images.
The second category is *Quantization Aware Training* (QAT). As previously discussed, quantization may shift the model away from its optimal convergence point, leading to reduced inference accuracy. To counteract this phenomenon, QAT attempts to emulate quantized inference during training, while the actual training still occurs in floating point.
Quantization is modeled during training by inserting fake quantization nodes for both weights and activations. Backpropagation is performed in floating point, as accumulating gradients in quantized precision can result in diminishing gradients or high errors, especially in low-precision scenarios. In fact, the weights adapt to the loss of precision caused by quantization.
This process can be computationally expensive, as it requires retraining the model for multiple epochs. Additionally, it may be sensitive to training hyperparameters, including S and Z.

Chapter 3

Tools and Methodology

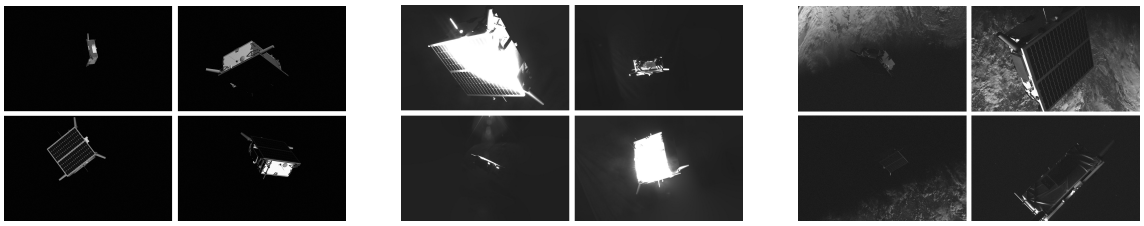
This chapter is divided into two main sections. The 'Tools' section presents the Python libraries, frameworks, and inference engines used to develop a real-time, vision-based object detection system that operates on the dataset presented herein. Finally, a technical description of the Jetson Orin Nano embedded hardware employed to test the model is given.
The second section, 'Methodology', describes the methodologies adopted, starting from image processing and data augmentation, and arriving at model training, testing, model format conversion, and quantization.

3.1 Tools

3.1.1 Dataset

In ML, the term dataset refers to a collection of data used to train and test algorithms and models. In this thesis, the dataset used is the PosE Estimation Dataset + (SPEED+), developed by Stanford University and utilized in the second International Satellite Pose Estimation Challenge, co-hosted by SLAB and the Advanced Concepts Team of the European Space Agency. This dataset is designed to evaluate and compare the robustness of space-borne ML models trained on synthetic images [46].
The dataset is composed of images of Tango, an advanced and highly maneuverable spacecraft part of the PRISMA mission. This mission serves as a technology demonstration for the in-flight validation of sensor technologies and guidance/navigation strategies for spacecraft formation flying and rendezvous [17].
Acquiring a large-scale labeled dataset of images of an intended target in a space environment is complex due to the limited memory and power onboard a spacecraft, as well as the difficulty of transmitting large volumes of data to Earth. For this reason, the majority of datasets rely on synthetic images, which are relatively less challenging to produce. However, these synthetic images often fail to fully replicate the visual features and illumination variability inherent in real spaceborne images.
The novelty of SPEED+ lies in the fact that, in addition to 59,960 synthetic images, it also includes 9,531 Hardware-In-The-Loop (HIL) images, represented in figure 3.1. These HIL images are captured in a ground-based simulated space environment to

better approximate real-world scenarios.
In particular, the HIL images are divided into lightbox and sunlamp pictures. The following experimental setup is used to generate them. A lightweight, reduced-scale mockup of Tango is positioned with different attitudes and held by a robotic arm, while being exposed to varying lighting conditions. Meanwhile, another robotic arm, equipped with a camera, moves along a ceiling-mounted linear rail across the room to capture images from different perspectives.
The lightbox setup is designed and calibrated to provide a uniform maximum radiance of 14 $W/m^2 \cdot sr$, corresponding to the mean radiance from Earth for an albedo coefficient of 0.3 and a solar irradiance of 1366 W/m^2. The sunlamp setup, on the other hand, simulates direct sunlight exposure with a collimated beam at a solar constant of 1.0 (nominal 1357 W/m^2) and a spectral response close to 6000 K.
The HIL images are intended for the final evaluation of the ML algorithm with respect to domain gaps, enabling a more accurate assessment of its robustness under real-world conditions



(a) Synthetic (b) Sunlamp (c) Lightbox

Figure 3.1: SPEED+ synthetic and HIL images [46]

The synthetic images used for training and validation are stored in the same folder and named in the format imgxxxxxx.jpg, where 'x' represents an integer. The labels associated with each image are provided in two separate JSON files, one for training and one for validation, containing the quaternions and translations. The division between training and validation sets is made randomly. For example, the training labels do not include all the images but only a subset of the synthetic dataset, selected randomly. Therefore, it is necessary to link each image with its corresponding label, using the proper label and the image names.
Table 3.1 presents the specifics of the camera used to collect the pictures, which are needed during the keypoints extraction process.

3.1.2 Albumentations

When dealing with CV and ML algorithms, data augmentation is an essential technique that enhances the robustness of a model by increasing the volume of available training data.
Additionally, each model requires input data in a specific format, making preprocessing of the original images necessary.
It is possible to perform data augmentation and pre-processing using the Python library Albumentations [19]. This tool is particularly well-suited for data preparation in DL contexts, as it allows for the automatic application of the same transfor-

Table 3.1: Camera Parameters [46]

Parameter	**Description**	**Value**
N_u	Number of horizontal pixels	1920
N_v	Number of vertical pixels	1200
f_x	Horizontal focal length [m]	0.017513
f_y	Vertical focal length [m]	0.017513
p_x	Horizontal pixel length [μm]	5.86
p_y	Vertical pixel length [μm]	5.86
$[r_1, r_2, r_3]$	Radial distortion parameters	[-0.2238, 0.5141, -0.1312]
$[t_1, t_2]$	Tangential distortion parameters ($\times 10^{-4}$)	[-6.650, -2.140]

mations to both input images and their corresponding labels, such as keypoints, bounding boxes, and masks.
In image processing, there are mainly two types of transformations that can be applied. Spatial-level transformations modify the spatial arrangement of pixels, as illustrated in figure 3.2. Meanwhile, pixel level transformations act on the intensity, color, or position of pixels in an image without affecting its labels, as shown in figure 3.3.

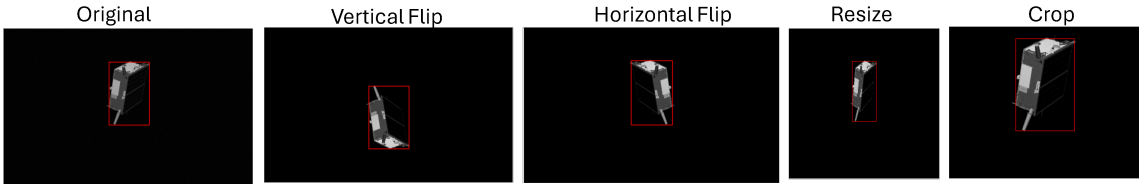


Figure 3.2: Spatial transformations

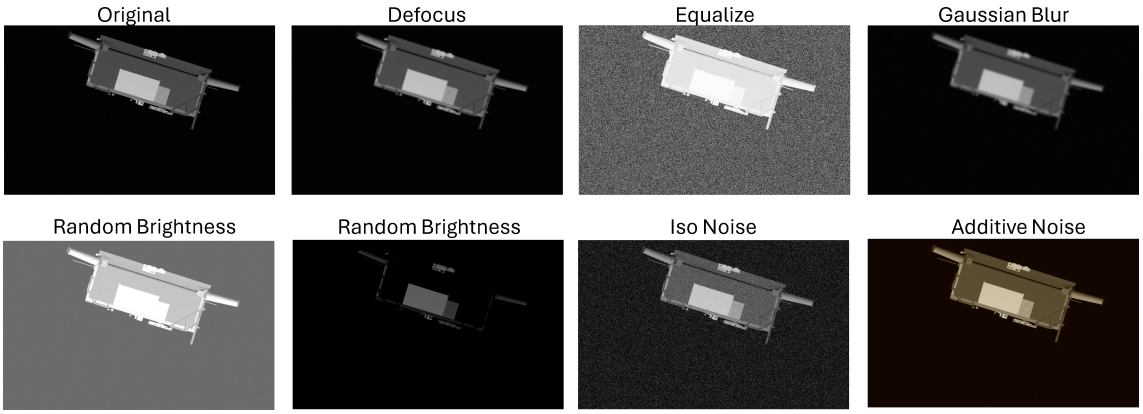


Figure 3.3: Pixel transformations

3.1.3 PyTorch

PyTorch is the most popular open-source DL framework among researchers and major companies like Tesla and Microsoft. As of January 31, 2025, more than 58% of deep learning research papers utilize PyTorch as shown in figure 3.4 [61].

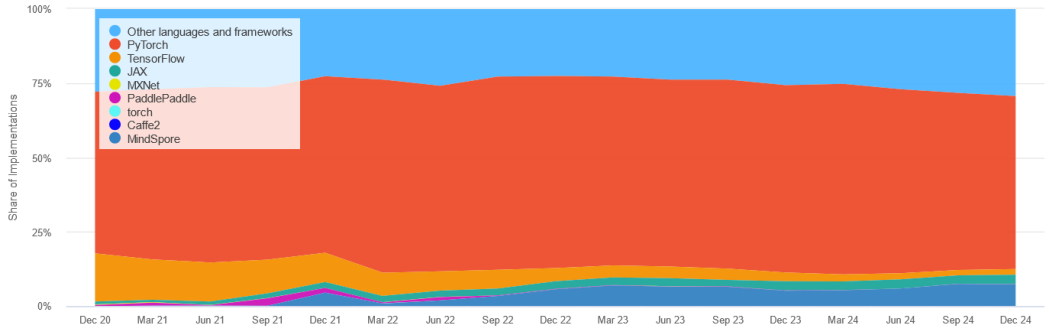


Figure 3.4: Paper implementations grouped by framework [61]

There are multiple reasons behind the widespread adoption of PyTorch in experimentation and prototyping. First of all its accessibility: being written in Python, it is approachable by the majority of ML practitioners. Another key factor is that it allows a straightforward access to layers and weights, making debugging more intuitive. Additionally, PyTorch employs reverse-mode auto-differentiation, which enables computation graphs to be modified on the fly [49].
At the core of PyTorch are tensors, a fundamental data type similar to multidimensional arrays, used to store and manipulate model inputs, outputs, and parameters. While similar to NumPy's array, PyTorch tensors have the additional capability of running on GPUs, significantly accelerating computations. This is possible thanks to native integration with CUDA, a programming model and computing toolkit developed by NVIDIA. CUDA, in particular, enables compute-intensive operations to run efficiently by parallelizing tasks across GPU cores. PyTorch provides built-in CUDA support via the *torch.cuda* module, allowing tensors and operations to be easily transferred between CPU and GPU memory. However, it is essential for the correct execution of operations that the versions of PyTorch and CUDA are compatible.
PyTorch is also highly efficient for complex operations such as quantization and parallelization, which, as described in the previous chapter, enhance inference speed.
Finally, another important feature is the export function, which allows exporting a PyTorch model into different formats, such as TorchScript, ideal for inference in production environments, and Open Neural Network Exchange (ONNX).
ONNX is an open standard designed to represent machine learning models, allowing them to be trained in one framework (such as PyTorch or TensorFlow) and then exported to run in another environment optimized for different hardware architectures. This enables better access to hardware-specific optimizations, ultimately improving inference speed and efficiency.

3.1.4 Inference engines

An Expert System (ES) is a system that, through logical rules, extracts knowledge from data captured to solve problems that ordinarily require human expertise [12]. An inference engine is the brain of an ES. Its main function is to draw inferences from data using a set of rules, applying them to a knowledge base to make decisions. It interprets data, derives new insights, and supports decision-making or predictions.
ONNX Runtime, developed by Microsoft, is a cross-platform DL model accelerator and a versatile inference engine often used in model deployments operations. Its flexible interface allows integration with hardware-specific libraries, often improving performance with respect to the original framework [25].
ONNX Runtime, during an inference process, first applies a series of graph optimizations to the model graph and subsequently partitions the optimized main graph into subgraphs based on available hardware-specific accelerators.
Optimized computation kernels in the core ONNX Runtime provide performance improvements, and assigned subgraphs benefit from further acceleration by each Execution Provider (EP). ONNX Runtime abstracts away the complexities of hardware libraries, which are crucial for optimizing DNN execution across diverse platforms like CPUs and GPUs. It achieves this by enabling EPs to allocate specific nodes or subgraphs for execution using optimized libraries on supported hardware platforms. Major providers include CUDA, TensorRT, and others.
NVIDIA TensorRT is an ecosystem of APIs for high-performance DL inference. TensorRT includes an inference runtime and model optimizations that deliver low latency and high throughput for production applications [24]. Built on the CUDA parallel programming model, it optimizes inference using techniques such as quantization, layer and tensor fusion, and kernel tuning on all types of NVIDIA GPUs, from edge devices to PCs to data centers. It powers key NVIDIA solutions such JetPack.
Another key feature of ONNX Runtime is the ability to implement both PQT and QAT model quantization.
In particular, for PQT quantization, both dynamic and static quantization are available.
Dynamic quantization computes the quantization parameters, such as scale S and zero point Z, for activations dynamically. This means that those parameters are specific to each forward pass, guaranteeing higher accuracy.
On the other hand, static quantization first runs the model using a set of inputs called calibration data. During these runs, the quantization parameters for each activation are computed. These quantization parameters are written as constants into the quantized model and used for all inputs. Having all the activations with the same S and Z during each forward pass will reduce the computational cost at the expense of reduced accuracy.
It is possible to quantize the model in two different representations. In the Operator-oriented (QOperator) representation, all the quantized operators have their own ONNX definitions, such as QLinearConv or MatMulInteger. The Tensor-oriented (QDQ) representation inserts operations like *DeQuantize* and *Quantize* layers between the original operators to simulate the process of quantizing and then restoring the tensor values. This ensures that the tensor's data is efficiently processed while

maintaining its structure. An example of the different representations is given in figure 3.8.

3.1.5 Jetson Orin Nano

NVIDIA Jetson is a series of embedded computing boards designed to accelerate ML applications. These compact and powerful devices, built around NVIDIA's GPU architecture, can run complex AI algorithms and DL models directly on the device. For this reason, Jetson boards have found applications in many different fields, including robotics, autonomous vehicles, and industrial automation, where AI inference needs to be performed locally with low latency and high efficiency. Additionally, these boards are based on the ARM architecture CPU and run on lower power compared to traditional GPU computing devices [35].
NVIDIA Jetson is powered by the Jetpack software development kit (SDK). This SDK includes the Linux for Tegra (L4T) operating system and the CUDA Accelerated AI stack, with a complete set of libraries for GPU computing acceleration, multimedia, graphics, and computer vision. In addition, Jetpack provides a collection of ready-to-use services that accelerate AI application development on Jetson [23].
Jetson Orin Nano, in particular, offers a promising combination of a multi-core ARM A78AE CPU (up to 6 cores with a frequency of 1.5 GHz) and an NVIDIA GPU (8 SMs, frequency up to 0.625 GHz), providing sufficient computing power for the implementation of AI algorithms and the processing of complex sensor data. Consuming only 15W of maximum power, Orin Nano is an energy-efficient alternative for power-constrained applications. More specific technical details are presented in table 3.2.
As previously stated, modern space missions require high payload processing performance. A growing trend is the direct extraction of useful data on-board, which must be guaranteed with minimal latency. Unfortunately, current families of processors often fall short of meeting these evolving performance requirements. For this reason, Jetson Orin Nano, with its specific features, is a good candidate to meet new and increasingly demanding requirements [51]. Jetson Orin Nano results well suited for space applications, also for its high radiation tolerance. The study carried out in [51] demonstrates that this device, along with Jetson Orin NX and Xavier NX, can survive total ionizing doses (TID) of at least 20 krad. This is a significant value for missions in LEO. Although the study highlights a performance degradation (CPU clock frequency reduction) at higher doses of 50 krad, the demonstrated resilience makes it a promising candidate for space applications.
It is also important to consider the involvement of external components to guarantee the reliability of this device. For example, specialized porting boards that shield against radiation can mitigate the risk of premature failures due to radiation exposure [51].
The Jetson Orin Nano used in this thesis is powered by Jetpack 6.2.

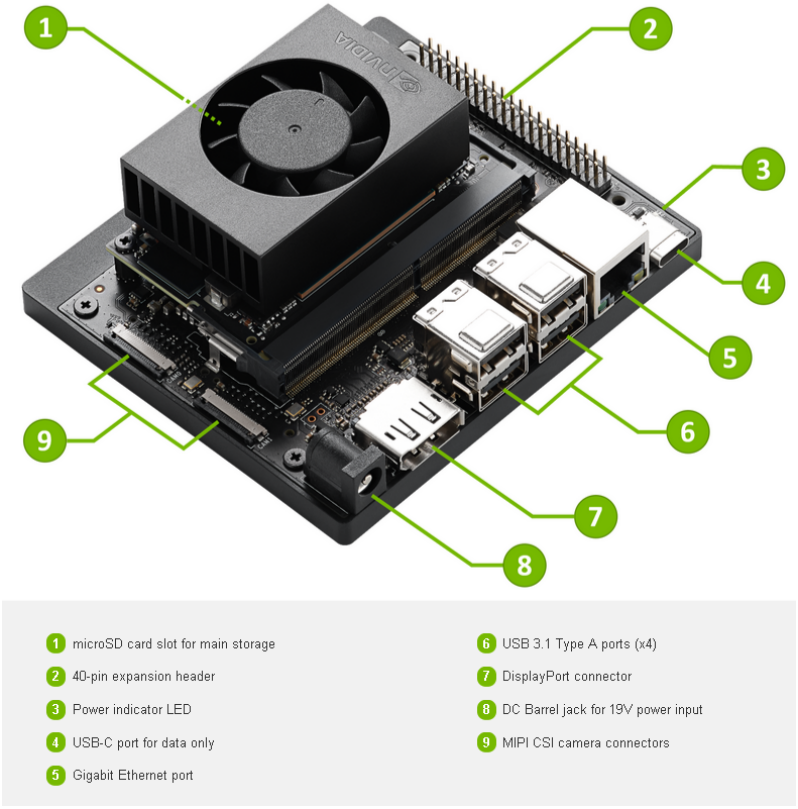


Figure 3.5: Jetson Orin Nano [35]

3.1.6 Ultralytics

Ultralytics is an open-source Python library developed by the American software company with the same name, which specialises in CV and DL. Ultralytics allows, through the YOLO family of algorithms, to perform classification, object detection, and image segmentation with a straightforward procedure [62]. The library supports a wide range of operations, including training and validation on custom datasets, testing and model deployment, as well as data augmentation and model quantization. Another interesting feature is the exporting capability in ONNX and TensorRT format, allowing for better inference performance.
It is important, in particular during the training and validation processes, that the labels are in YOLO format, described by equation 3.1:

$$[< class - id >, < x_{center} >, < y_{center} >, < width >, < height >] \qquad (3.1)$$

where $class - id$ is the class label of the object, represented by a number; x_{center} and y_{center} are the x and y coordinates of the object's center, normalized by the image width and height, respectively; and $width$ and $height$ are the width and height of the bounding box, normalized in the same way as the bounding box center coordinates. Each image in the dataset must have a corresponding *.txt* file, named as the image, containing one row per object with the BB specific. The dataset has to be organized into three main folders: 'train', 'val', and 'test', each containing 'images' and 'labels' subfolders. A *.yaml* file specifying the paths to these folders is required during training.

Table 3.2: Main Specifications of NVIDIA Jetson Orin Nano [35]

Feature	**Details**
GPU	NVIDIA Ampere, 1024 CUDA cores, 32 Tensor Cores
CPU	6-core ARM Cortex-A78AE
RAM	8 GB LPDDR5
Storage	microSD (expandable via M.2 NVMe SSD)
AI Performance	Up to 40 TOPS (Tera Operations Per Second)
Power Consumption	7W / 15W (configurable)
Interfaces	1x GbE, 3x USB 3.2, 1x USB 2.0, GPIO, I2C, I2S, SPI, UART
Video	Encode/Decode 4K60, H.264/H.265 support
Operating System	Ubuntu-based NVIDIA JetPack SDK
Size	100 x 79 x 21 [mm]

During inference, Ultralytics YOLO provides various metrics, such as precision, recall, mAP, and inference time, to evaluate model performance.

3.2 Methodology

3.2.1 Data collection

The first step done in this thesis project was the selection and collection of data. In particular, the training dataset is composed of 10.000 images taken from the synthetic images corresponding to the first 10.000 labels in the training JSON file. This size was considered sufficient, considering the trade-off between training time, variety of subjects and achievable accuracy. The main goal of this thesis is, in fact, optimizing inference speed, rather than achieving the best detection accuracy, which could be obtained by using the entire dataset. Nevertheless, this number proved to be large enough to achieve reasonable accuracy.
As already described, the labels in the training JSON file correspond to a subset of all the synthetic images. For this reason, it was necessary to match the labels with the images that have the same name. Subsequently the same operation was conducted for the first thousand labels of the validation json file. Regarding the test set, of 1000 synthetic images, it was sufficient to select the first images since the order of the image names was the same in the label JSON file and in the test image folder. Additionally, the same operation was applied to the HIL images to compare the robustness of the developed algorithm with that of YOLO.
The second step was the extraction of the pixel coordinates of the keypoints of the Tango S/C. These were obtained using the 3D coordinates of 11 vertices of the Tango spacecraft, along with the camera characteristics (shown in table 3.1), the quaternions q and the vector translations t. The procedure was performed thanks to a given script. The steps followed in the procedure are as follows.

From the vector of quaternions q it is possible to define the rotational matrix R as described in equation 3.2.

$$R = \begin{bmatrix} 2q_0^2 - 1 + 2q_1^2 & 2q_1q_2 + 2q_0q_3 & 2q_1q_3 - 2q_0q_2 \\ 2q_1q_2 - 2q_0q_3 & 2q_0^2 - 1 + 2q_2^2 & 2q_2q_3 + 2q_0q_1 \\ 2q_1q_3 + 2q_0q_2 & 2q_2q_3 - 2q_0q_1 & 2q_0^2 - 1 + 2q_3^2 \end{bmatrix} \tag{3.2}$$

Thanks to R it is then possible to transform, the 11 points, from the *satellite* to the *camera* reference system with equation 3.3.

$$\begin{bmatrix} X_{camera} \\ Y_{camera} \\ Z_{camera} \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_{satellite} \\ Y_{satellite} \\ Z_{satellite} \\ 1 \end{bmatrix} \tag{3.3}$$

The next step was to project the 3D points into the 2D image plane, by applying the perspective projection from equation 3.4.

$$\begin{aligned} x' &= \frac{X_{camera}}{Z_{camera}} \\ y' &= \frac{Y_{camera}}{Z_{camera}} \end{aligned} \tag{3.4}$$

When processing a digital image, it is necessary to introduce a correction for distortions that can cause a degradation in the quality of the final image. In particular, in this work, corrections were introduced for both radial and tangential distortion. Radial distortion causes straight lines to bend as general curves, and points are moved in the radial direction from their correct position. Tangential distortion arises from positional defects, i.e., eccentricity of the optical axis or the lack of parallelism of individual lenses with respect to each other, but also with respect to the photographic sensor of the camera [28].
Here a polynomial distortion correction was adopted, such as the one described in equation 3.5.

$$\begin{aligned} d^2 &= x'^2 + y'^2 \\ c_{dist} &= 1 + r_1d^2 + r_2d^4 + r_3d^6 \\ x'' &= x'c_{dist} + 2t_1x'y' + t_2(d^2 + 2x'^2) \\ y'' &= y'c_{dist} + t_1(d^2 + 2y'^2) + 2t_2x'y' \end{aligned} \tag{3.5}$$

Where the r_i coefficients, along with the squared distance d from the center of the image, are used to define the scale factor c_{dist} for radial corrections, while the t_i coefficients account for the tangential distortion correction. The values considered are the ones described in table 3.1.
Finally, the 2D coordinates in pixel values of the key points are obtained through matrix multiplication with the camera matrix and the corrected coordinates, as shown in equation 3.6.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{3.6}$$

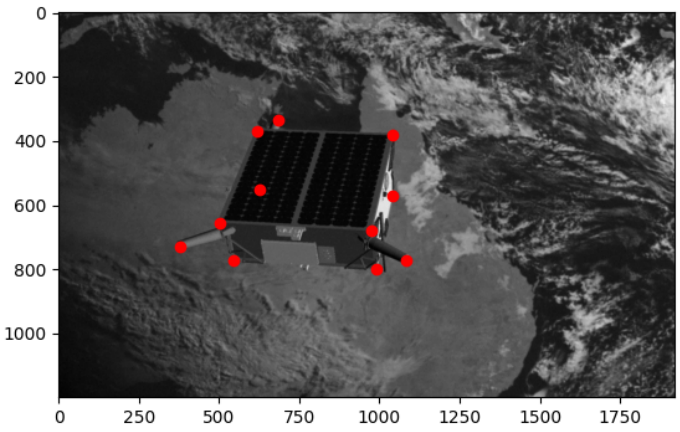


Figure 3.6: Projection of the 11 3D keypoints on a 2D image

The values in the camera matrix represent f_x and f_y, the focal lengths in pixel units, while c_x and c_y are the pixel coordinates of the image center, i.e., the point where the Z_{camera} axis intersects with the image plane.
Each keypoint's coordinate is stored in the typical (x, y) format and allocated in a common array.
An example of the results of the projection of the keypoints onto an image is shown in figure 3.6.
At this point, in order to build a bounding box, a Python function was implemented to extract the coordinates of the top-left and bottom-right corners from the 11 keypoints. The function performs the operations described in equation 3.7, which give the final label in Pascal VOC format $[x_{min}, y_{min}, x_{max}, y_{max}]$.

$$\begin{aligned} x_{min} &= min(x_{kpts}) \\ y_{min} &= min(x_{kpts}) \\ x_{max} &= max(x_{kpts}) \\ y_{max} &= max(y_{kpts}) \end{aligned} \tag{3.7}$$

A second function was implemented to transform the label from Pascal VOC format to YOLO format, applying the operations described in equation 3.8, where W and H are the width and height of the image.

$$\begin{aligned} x_{center} &= \frac{x_{min} + x_{max}}{2W} \\ y_{center} &= \frac{y_{min} + y_{max}}{2H} \\ widht &= \frac{x_{max} - x_{min}}{W} \\ height &= \frac{y_{max} - y_{min}}{H} \end{aligned} \tag{3.8}$$

Additionally a class tag, 0, was added in order to define the class "Tango".

3.2.2 Image transformations

The EfficientNet-B0 requires normalized input tensors. Furthermore, to increase robustness and avoid overfitting problems, as already mentioned above, data augmentation is recommended during the training process. For these reasons, two different image transformations, one for training and one for validation and testing, are implemented using the Albumentations library. The training transformation is described in table 3.3. Regarding the validation transformations, only the *Resize*, *Normalize* and *ToTensor* operations are considered.

Table 3.3: Training Transformations

Transformation	**Probability**	**Description**
Resize	-	Resizes the image to 224x224 pixels.
Rotate	0.8	Rotates the image randomly up to 15 degrees with a constant border mode.
RandomBrightnessContrast	0.2	Adjusts brightness and contrast randomly with a limit of 0.5.
OneOf (Set 1)	1.0	Applies one of the following transformations with equal probability:
- GaussNoise	0.8	Adds Gaussian noise to the image.
- CLAHE	0.8	Enhances local contrast using Contrast Limited Adaptive Histogram Equalization.
- ImageCompression	0.8	Simulates image compression artifacts.
- RandomGamma	0.8	Randomly adjusts the gamma of the image.
- Posterize	0.8	Reduces the number of colors in the image.
- Blur	0.8	Applies a blurring effect to the image.
OneOf (Set 2)	1.0	Reapplies one of the transformations from the previous set with the same probabilities.
Affine	0.2	Applies affine transformations with a translation of 10%, scaling between 0.9 and 1.1, and no rotation, using a constant border mode.
Normalize	-	Normalizes the image using mean = [0.382, 0.382, 0.382] and std = [0.382, 0.382, 0.382], with max pixel value of 255.
ToTensorV2	-	Converts the image to a tensor format.

An example of images that have undergone the training transformation is presented in figure 3.7.
The YOLOv8n from Ultralytics automatically resizes the image to 640x640 pixels if the image is square. In the case that the image is larger than the required dimension and not squared, Ultralytics will resize the larger dimension to 640 and then scale the other one maintaining a constant aspect ratio. The remaining space is then padded to achieve a square 640x640 input. Being the images from SPEED+ 1920x1200 pixels the input of the YOLO model are resized at first to 640x416 pixels and then padded to 640x640 pixels.
Furthermore, during the training process, data augmentation is applied by default, using the transformations shown in the table 3.4.

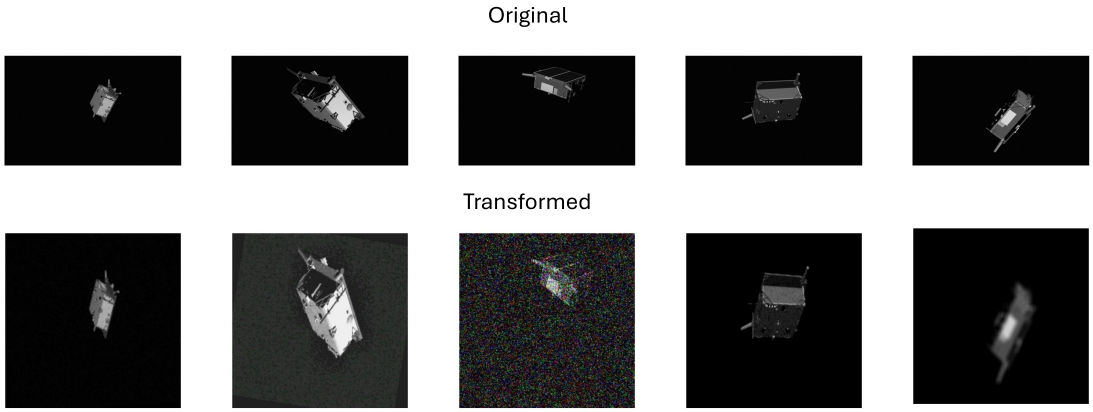


Figure 3.7: Image transformation for the training process

3.2.3 PyTorch training pipeline

In the following, the procedures carried out in PyTorch to ensure the proper training of the modified EfficientNet-B0 model are described.
The first step was building a custom *Dataset* class to store the samples and their corresponding labels. This class must have three main functions: *__init__*, *__len__* and *__getitem__*. The task of the class dataset, here named *BoundingBoxDataset*, is to initialize the directory with the images, annotations, and transforms, be able to return the number of samples in the dataset, and, finally, load, transform, and return a sample from the dataset at a given index.
Once the data are stored in *BoundingBoxDataset*, it is possible to access one sample with its corresponding label at a time. For this reason, in order to access multiple samples at a time, the dataset is passed through a *DataLoader* that loads the images and labels in batches.
In particular, two *Dataset*s and two *DataLoader*s were built, allowing the application of two different transformations to the original data, depending on whether they are meant for the training or validation process. Furthermore, for the training *DataLoader*, the shuffle option is enabled to shuffle the data once a forward pass of all the batches is completed.
As already stated above, the selected model is the EfficientNet-B0. However, since this model is designed for image classification, it was necessary to modify its last layer. This layer, in fact, is a fully connected layer, whose task is to take the feature map generated by the previous convolutional layers and predict the class. To adapt the model, this layer was replaced with a linear layer that takes 1280 input features and produces 4 outputs, which correspond to the coordinates of the desired points.
Furthermore, two main functions for training, *train_one_epoch*, and for validation, *val_one_epoch*, over one epoch were defined.
Considering the model being already on the GPU, *train_one_epoch* implements the following procedure to allow the cost function to converge toward a minimum:

1. *.to(device)*: Allocate image and label tensors on the GPU.
2. *Forward pass*: Make predictions with the model.

Table 3.4: YOLO Training default data augmentation parameters [62]

Argument	Type	Default	Description
hsv_h	float	0.015	Adjusts the hue of the image by a fraction of the color wheel, introducing color variability. Helps the model generalize across different lighting conditions.
hsv_s	float	0.7	Alters the saturation of the image by a fraction, affecting the intensity of colors. Useful for simulating different environmental conditions.
hsv_v	float	0.4	Modifies the value (brightness) of the image by a fraction, helping the model perform well under various lighting conditions.
translate	float	0.1	Translates the image horizontally and vertically by a fraction of the image size, aiding in learning to detect partially visible objects.
scale	float	0.5	Scales the image by a gain factor, simulating objects at different distances from the camera.
fliplr	float	0.5	Flips the image left to right with the specified probability, useful for learning symmetrical objects and increasing dataset diversity.
mosaic	float	1.0	Combines four training images into one, simulating different scene compositions and object interactions. Highly effective for complex scene understanding.
erasing	float	0.4	Randomly erases a portion of the image during classification training, encouraging the model to focus on less obvious features for recognition.
crop_fraction	float	1.0	Crops the classification image to a fraction of its size to emphasize central features and adapt to object scales, reducing background distractions.

3. *Calculate the loss*: Evaluate the loss between the predictions and the ground truth labels.

4. *.zero_grad()*: This is the first step of backpropagation. To avoid accumulating gradients from previous steps, set them to zero.

5. *.backward()*: Backpropagate the loss to compute the gradients of each model parameter with respect to the loss.

6. *.step()*: Perform gradient descent, updating the weights through the optimizer function using the newly computed gradients.

For *val_one_epoch*, fewer steps are necessary since there is no backpropagation. Additionally, the model was set in *.eval()* mode. This operation, along with *torch.no_grad()*, turns off specific layers and parts of the model needed only during training, allowing for faster and less computationally demanding execution. To conclude just point 1., 2. and 3. are performed in this function.
Table 3.5 shows the selected hyperparameters for the training and validation loop.

Table 3.5: Training hyperparameters

HYPERPARAMETERS	
EPOCHS	100
BATCH	16
Loss Function	MSE
OPTIMIZER	ADAM
Learning Rate	$1e^{-3}$
Weight Decay	$5e^{-4}$

3.2.4 PyTorch Inferences

To perform inference with PyTorch, the function *test_fn* was implemented. Here, the operations conducted are similar to those in *val_one_epoch*. The main difference is that pre-processing is now performed without the use of a *DataLoader*, using the *pre_process* function, where the same transformations from the validation steps are implemented. This is done to simulate a real-world scenario, where neither a *Dataset* nor a *DataLoader* is present.
Additionally, the *post_process* function was implemented to extract the IoU from the predictions. Here, two main steps are carried out. In the first step, a conversion of data is performed. The predictions are made for a 224x224 pixel image. For this reason, the predictions are scaled back to the original dimensions of the image. Then, the IoU is evaluated by comparing the scaled predictions with the ground truth labels.
The IoU is evaluated using a dedicated function. The first step is to determine the coordinates of the corners of the rectangle representing the intersection area, using the formulas in equation 3.9. From where, it is straightforward to evaluate the intersection area.

$$\begin{aligned} x_{min}^{intersection} &= \max(x_{min}^{pred}, x_{min}^{gt}) \\ y_{min}^{intersection} &= \max(y_{min}^{pred}, y_{min}^{gt}) \\ x_{max}^{intersection} &= \min(x_{max}^{pred}, x_{max}^{gt}) \\ y_{max}^{intersection} &= \min(y_{max}^{pred}, y_{max}^{gt}) \end{aligned} \tag{3.9}$$

The union area is evaluated with equation 3.10.

$$\text{Union Area} = \text{Area}_{\text{pred}} + \text{Area}_{\text{gt}} - \text{Area}_{\text{overlap}} \tag{3.10}$$

In case of no intersection, the IoU is set to zero, skipping the other steps to avoid computational issues, such as division by zero. During these steps, different timers were set to record the inference, pre-processing and post-processing times.

3.2.5 ONNX Runtime pipeline

Once the PyTorch model was trained, thanks to the *torch.export* function, it was possible to export it in ONNX format. During the export, the parameters shown in table 3.6 were set, with, in particular, the dynamic axes were enabled to allow the model to accept more than one batch at a time as input.

Table 3.6: ONNX exportation settings

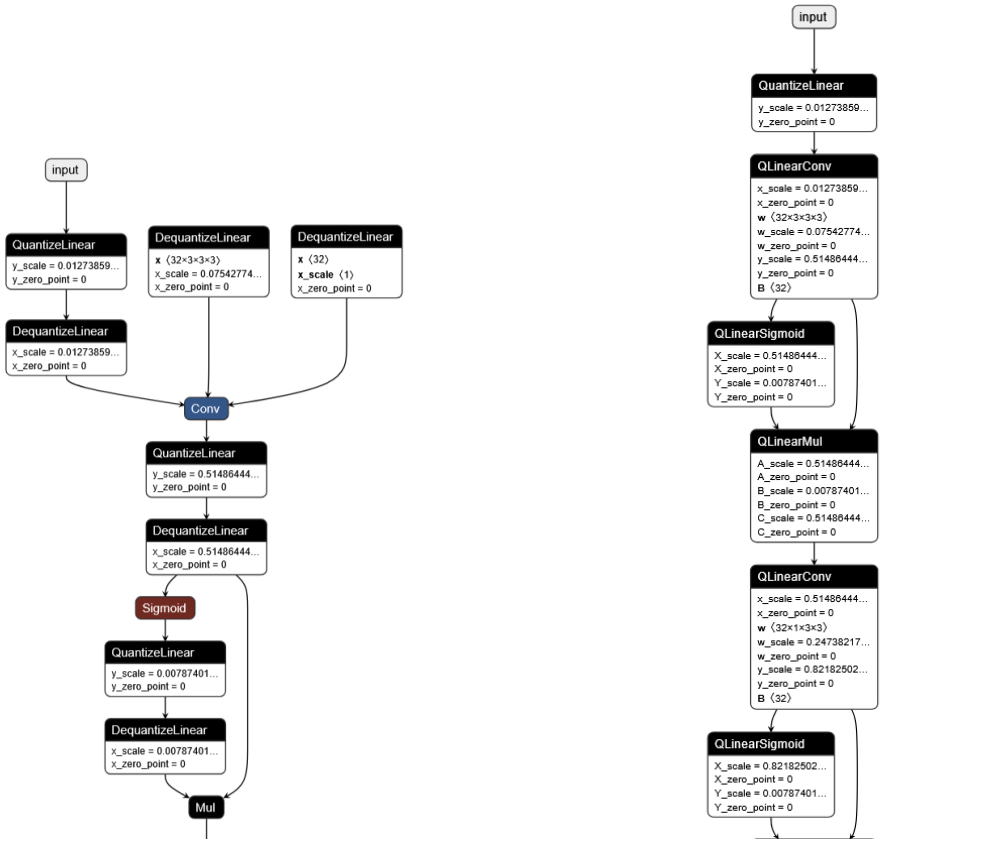
SETTINGS	
input shape	(1, 3, 224, 224)
opset version	11
input names	['input']
output names	['output']
dynamic axes	{'input': {0: 'batch size'}, 'output': {0: 'batch size'}}
export parameters	True

In order to perform inference with an ONNX model, it is necessary to start an ONNX Runtime inference session, where the model path and the providers must be specified. As with PyTorch inference, *pre_process* and *post_process* functions were also used to perform inferences.
Here, in particular, the *pre_process* function was adapted to handle batch sizes different from one, in order to test the variations in speed performance for increasing batch sizes.
Finally, model quantization was carried out to achieve higher performance in terms of inference speed and FPS.
In particular, for CNNs a static quantization is recommended. For this reason a calibration dataset was defined. The Python class *CalibrationDataSet*, that implements the *__init__*, *get_next* and *rewind* functions, was defined. This class loads the calibration dataset, which is the entire testing dataset, and divides it into batches. The images in the batch are transformed using the validation transform function. Then, with *get_next*, after a forward pass, the subsequent batch was transformed and passed to the model. Finally, *rewind* returns the class to the beginning of the batch sequence.
The first step to quantize the model was to pre-process it with a built-in function *quant_pre_proces*, allowing the quantization process to be conducted efficiently. The main pre-processing operations carried out are:

- Shape Inference: Determine the dimensions of all the tensors in the model.

- Optimization: Removes inefficiencies in the model to make it faster. Such operations include, for example, Operator Fusion, Dead Code Elimination (i.e., eliminating parts of the model that are not in use), Constant Folding, and Graph Simplification. These steps reorganize the model to reduce the number of operations or the complexity of the data flow.
- Saving: Save the optimized model.

Once the preprocessing was completed, it was possible to proceed with the quantization and saving of the model, setting the symmetric activations option in order to enable the use of the TensorRT provider.
Additionally, both QDQ and QOperator quantization representations were obtained, since inference with TensorRT providers encounters problems with the QDQ representation. An example of the final quantized model is shown in figure 3.8, where the first layers of the model are displayed with both quantization representations



(a) QOQ representation (b) QOperator representation

Figure 3.8: EfficientNet-B0 QDQ and QOperator quantized model representation

3.2.6 Ultralytics pipeline

Training with Ultralytics doesn't require any pre- or post-processing operations, as these are autonomously handled by the library. Once the .yaml file with the paths

to the training and validation folders is defined and provided to the model, along with the batch size, training begins. In particular, the hyperparameters involved are those shown in table 3.7. Most of the hyperparameters are left at their default values, such as the combination of different loss functions.

Table 3.7: YOLOv8 Default Training Hyperparameters

HYPERPARAMETERS	
EPOCHS	100
BATCH	16
Learning Rate	$1e^{-2}$
Weight Decay	$5e^{-4}$
Class Loss	BCEWithLogitsLoss
Box Loss	DFL + CIoU
Differentian Focal Loss	DFL
OPTIMIZER	SGD with Momentum

Regarding inferences, it was necessary to implement a post-processing function in order to be able to evaluate the IoU. This will increase the post-process time. No timers are required, since the inference, pre- and post-processing speed are intrinsically evaluated. This high degree of process management autonomy by the library, on one hand, speeds up the process, but on the other hand, it does not allow for much flexibility. In this way, for example, the post-processing time includes various operations and not just the evaluation of the IoU. This makes it impossible to make a direct comparison with the post-processing conducted with PyTorch.
With Ultralytics, it is possible to simply export the models in ONNX and .engine format by specifying the specific format in the *export* command.
The inferences in this format are done in the same way as the PyTorch model, i.e., by simply giving the path of the image to the model. In order to do inference with a batch of images, the variable *dynamic* has to be set to true.
The YOLO models from Ultralytics accept images in various formats, including paths, numpy array, .jpg, and others. The choice of giving only the path was made to allow the model to handle the pre-processing entirely, in order to compare the time needed when compared to the pre-processing defined above.

Chapter 4

Results

This chapter first presents the results of training both the EfficientNet-B0 and YOLOv8n models, comparing the obtained metrics where possible. Then, the inference results of both models with Torch, ONNX and TensorRT formats and fp32, fp16 and int8 precision on the Jetson Orin Nano are presented. Finally, related work on real mission space debris classification algorithms and the use of Jetson for space applications is discussed to validate the work done.

4.1 Training results

By leveraging the functions described in the previous chapter, the modified EfficientNet-B0 model was trained and validated on the selected portion of SPEED+. In particular, the best weights were saved from the 89th epoch, where a minimum validation RMSE of 4.94 was recorded. It should be noted that this epoch does not correspond to the maximum validation mAP values. These occur at epoch 90, with an mAP@95 of 0.721 (table 4.1).

Table 4.1: Best EfficientNet-B0 Validation Loss and Accuracy

	EPOCH	RMSE	mAP@50:95	mAP@50
Best Loss	89	4.940	0.673	1.00
Best Accuracy	90	5.161	0.721	1.00

Regarding the training of the YOLOv8n model, the highest mAP@95 value was obtained at epoch 97, reaching 0.99407. The epochs corresponding to the minimum for each loss and the best metrics are shown in table 4.2.
Figures 4.1 and 4.2 show the training and validation loss trends for both models. From these, it is already possible to confirm that the training process was successful, as the two loss curves do not diverge or reach a plateau at a high loss value, indicating that neither overfitting nor underfitting occurred.
Figure 4.3 presents a comparison of mAP@50 and mAP@50:95 achieved during validation for the two models. It is evident that the YOLOv8n model has superior

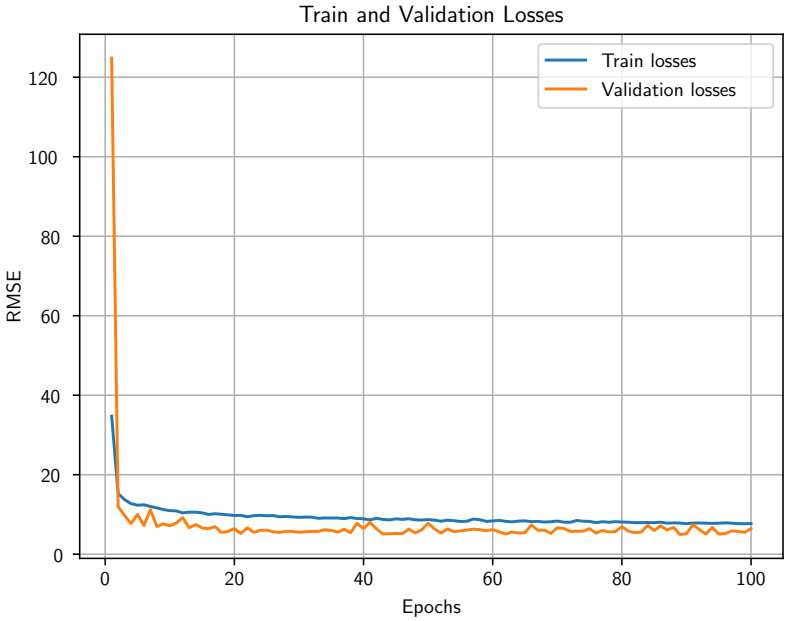


Figure 4.1: Training and Validation loss EfficientNet-B0

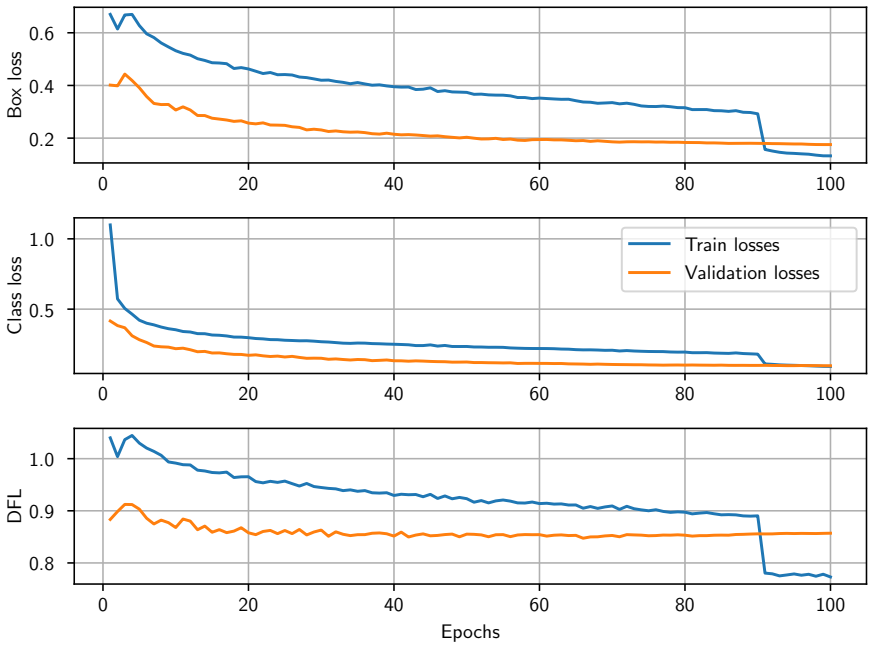


Figure 4.2: Training and Validation loss YOLOv8n

Table 4.2: Best YOLOv8n Validation Loss and Accuracy

	EPOCH	mAP@50:95	mAP@50	Box Loss	Class Loss	DFL
Best mAP@50:95	98	0.99407	0.995000	0.17571	0.09834	0.85640
Best Box Loss	99	0.99407	0.995000	0.17550	0.09769	0.85674
Best Class Loss	100	0.99406	0.995000	0.17550	0.09738	0.85703
Best DFL	66	0.99347	0.995000	0.19086	0.10929	0.84738

detection capability in terms of accuracy, achieving an mAP@50:95 that is 0.274 points higher than the EfficientNet-B0 model. However, for mAP@50, both models exhibit a similar trend. Although YOLOv8n converges earlier, by the end of training, both algorithms reach a mAP@50 of approximately 1.

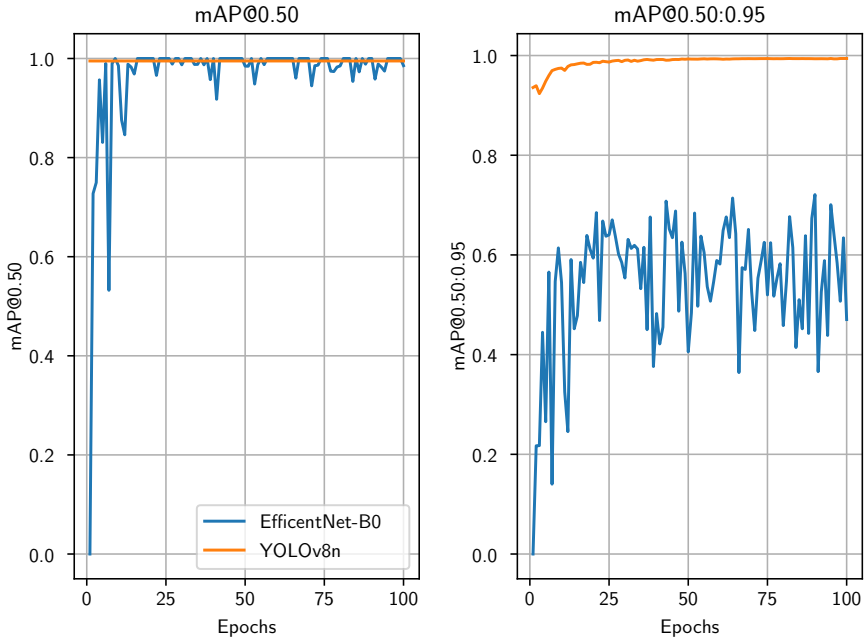


Figure 4.3: Comparison of validation mAP between EfficientNet-B0 and YOLOv8n

4.2 Inferences results

4.2.1 Accuracy results

Once the training was completed, it was possible to perform inference and compare the results obtained from the two models.
In particular, the algorithms were tested on both synthetic and HIL images to assess their robustness against the domain gap. The results obtained in terms of accuracy are listed in table 4.3.

Table 4.3: IoU results

	EfficientNet-B0	YOLOv8n
Synthetic	0.849	0.946
Sun lamp	0.383	0.442
Light box	0.381	0.442

As expected, YOLOv8n performs overall better; nevertheless, the EfficientNet-B0 model achieves relatively high accuracy, especially on synthetic images.
A visual representation of the detection inference results is provided in figure 4.4, where both ground truth and predicted BB are shown for both models.
Regarding the HIL images, both algorithms suffer significantly from the domain gap. Nevertheless, although the EfficientNet-B0 IoU is lower, it experiences almost the same percentage drop in accuracy, i.e., 55.09% for EfficientNet and 53.28% for YOLO.
It is evident that further strategies need to be applied to mitigate this performance drop and that data augmentation techniques alone are not sufficient.
Finally, a visual representation of the detection results for HIL images is presented in figure 4.5. In particular, figure 4.4b shows the results for EfficientNet-B0, while figure 4.5b presents the results for YOLOv8n.

4.2.2 Inference speed results

In order to assess the possibility of a true deployment of the detection algorithm, inference times were evaluated on the Jetson Orin Nano platform.
Different model formats were studied to obtain the most performant one in terms of inference time and FPS, without excessively compromising accuracy.
Table 4.4 presents the results obtained with EfficientNet-B0, while table 4.5 shows the results obtained from the YOLO inferences.
In the following, a discussion and interpretation of the values shown in tables 4.4 and 4.5 are provided.
At first, it is immediately noticeable how the effect of GPU acceleration, tested only on the EfficientNet-B0 PyTorch model with fp32 precision, guarantees more than a 51-times speedup compared to the CPU.
Considering the pre-processing times, the strategy implemented here with Albumentations is slightly slower compared to the one implemented by Ultralytics, even

Table 4.4: EfficientNet-B0 Jetson Orin Nano inference results. The superscript denotes the quantization representation adopted. Specifically, 1 stands for QDQ and 2 stands for QOperator.

Format	Precision	Inference Speed (ms)	FPS	IoU	Pre Speed (ms)	Post Speed (ms)	# Image
PyTorch CPU	fp32	1841	0.543	0.845	27.11	0.110	1000
PyTorch GPU	fp32	35.546	28.13	0.845	16.021	0.346	1000
ONNX CPU	fp32	61.872	16.16	0.845	20.35	0.1302	1000
ONNX CUDA	fp32	21.081	47.43	0.849	16.59	0.1337	1000
	fp16	17.520	57.07	0.849	14.33	0.119	1000
	int8[1]	29.854	33.49	0.436	18.72	0.1242	1000
	int8[2]	24.075	41.53	0.436	18.824	0.122	1000
TensorRT	fp32	10.774	92.81	0.845	17.13	0.1174	950
	fp16	7.350	136.05	0.850	14.25	0.1061	950
	int8[1]	11.151	86.88	1.71×10^{-6}	16.435	0.148	950
	int8[2]	28.031	35.67	0.436	17.866	0.125	950
TensorRT Batch	fp32	4.365	229.09	0.850	17.95	0.1123	976

Table 4.5: YOLOv8n Jetson Orin Nano Inference Results

Format	Precision	Inference Speed (ms)	FPS	IoU	Pre Speed (ms)	Post Speed (ms)	# Image
PyTorch	fp32	26.684	37.47	0.9460	7.20	431.36	1000
ONNX CUDA	fp32	28.661	34.89	0.945	7.784	408.189	1000
	fp16	33.84	29.55	0.945	8.50	456.19	1000
TensorRT	fp32	20.42	48.97	0.9453	9.710	499.30	1000
	fp16	11.647	85.85	0.945	8.634	432.71	1000
	int8	7.74	129.19	0.886	7.028	343.18	1000
TensorRT Batch	fp32	7.085	141.14	0.946	5.804	282.86	1000

though the pre-processing time is of the same order of magnitude in both cases. An important observation is that the pre-processing time is almost invariant with respect to the model format and precision. As expected, the post-processing time is much higher in the YOLOv8n inference, as a lot more operations are carried out, and for this reason, a fair comparison of the results obtained is not possible.
With these considerations made, some unexpected outcomes are noticeable, particularly for inference with the EfficientNet-B0 model in the ONNX quantized format. During inference of the ONNX format with the CUDA provider, for both quantization representations, ONNX Runtime inserts many *Memcpy* nodes into the model graph. This operation involves continuous data transfers between the CPU and GPU, significantly slowing down the entire process, especially if these transfers are frequent or if the tensors are large.
The creation of these nodes occurs because some operations in the quantized model may not be supported by the CUDA provider of ONNX Runtime, and as a result, they are executed on the CPU. In particular, for the QDQ representation, the Dequantized Layers nodes cause these issues, while for the QOperator representation, it is the layers whose names start with "Q".

Even though some operations are executed on the CPU, this solution still ensures a faster inference speed compared to the PyTorch model but suffers from a considerable drop in IoU due to the quantization process.
The same problem occurs during inference of the ONNX quantized model with the TensorRT provider, where TensorRT fails to correctly interpret specific layers of the neural network.
In particular, with the QDQ representation, if some nodes are not correctly converted, the model may skip them, resulting in incorrect predictions or NaN values. For this reason, the results obtained with the QDQ quantization version are presented for completeness, even though they are not considered reliable. Further work is needed to achieve the required optimization.
Even for the QOperator representation, some issues persist. TensorRT fails to find the necessary *plugins* to support quantized operations such as QLinearMul, QLinearGlobalAveragePool, QLinearSigmoid, and QLinearConv.
This occurs because the TensorRT provider does not have optimized implementations for these specific operations. Consequently, the entire model cannot be fully accelerated by TensorRT, and some operations, similar to the case of CUDA providers, must be executed on the CPU, once again slowing down the inference process. However, in this case, the IoU remains the same as that obtained with the CUDA provider, indicating that the overall execution is functioning correctly.
Another important observation regarding TensorRT providers is that, in order to function properly, they require a warm-up process using a set of images. For this reason, during the evaluation of the metrics, 950 images were considered instead of 1000.
Regarding the results obtained for YOLOv8n shown in table 4.5, the same problem observed with EfficientNet-B0 arises during inference with ONNX using the CUDA provider, i.e. some nodes are allocated on the CPU. This continuous transfer of data between the CPU and GPU during inference, again, significantly slows down the entire process.
Despite some results being discarded, it is still possible to draw final conclusions. EfficientNet-B0 has proven to be competitive when compared to the state-of-the-art YOLOv8n. In particular, impressive inference speed performance is achieved with the model accelerated using TensorRT with fp16 precision, which outperforms the fastest version of YOLOv8n, quantized and accelerated by TensorRT, by 6.86 FPS, while maintaining only a 4.1% decrease in accuracy, a minimal difference between the two models.
Another technique experimented with here to speed up inference is batching, where batches of images (in this case, 16) are processed simultaneously. As shown, this approach can significantly increase inference speed and is widely used in various fields, such as surveillance systems and retail for inventory management and customer behavior analysis. However, this technique still has some limitations in real-time applications, as it may introduce overhead when loading the entire batch, require more memory, and cause latency issues since predictions are only available after the entire batch has been processed. Nevertheless, it can substantially speed up the inference process, and its use in this sector can bring significant benefits in terms of inference speed.

Finally, a visual interpretation of the results in terms of speed and accuracy is presented in figures 4.6 and 4.7, while a representation of the speedups achieved with different model formats is shown in figure 4.8. In particular, figures 4.6 and 4.7 only display results considered reliable and comparable. Since Ultralytics does not allow for direct quantization of the YOLO ONNX model, these results are not included in the plots.

4.2.3 Literature validation

A final comparison is made by considering significant literature examples.
In the aforementioned RemoveDEBRIS mission, a vision-based navigation system utilizing a digital camera and LiDAR was also tested for debris observation. This system automatically determines key parameters such as distance and spinning rates, which are essential for rendezvous and debris capture. As reported in [5], the processing time for each frame was less than 0.1 seconds, thereby strongly validating the inference time results obtained in this work.
In [41], a different strategy to reduce inference time is presented, based on image compression techniques rather than modifying the model itself. In particular, two object detection algorithms are tested: the Single Shot MultiBox Detector (SSD) and the Region-based Fully Convolutional Network (R-FCN), both pre-trained on the DOTA dataset [67], a large-scale dataset for object detection in aerial images. Inference speed results are reported for the smallest tested image (475×546 pixels) and the largest runnable one (4392×4441 pixels). Thanks to these techniques, significant improvements in inference speed are achieved on the Jetson Nano, as shown in Table 4.6.

Table 4.6: Inference speed obtained on Jetson Nano from [41] in seconds.

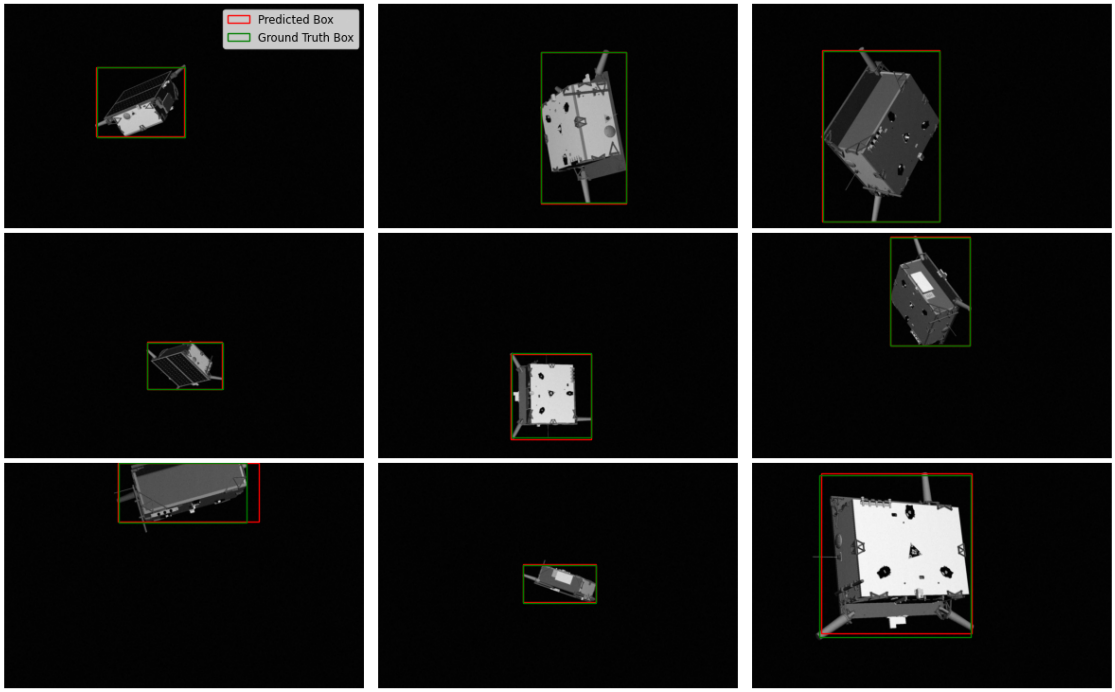
	Large image		Small image	
	Original	Compressed	Original	Compressed
SSD	5.21	3.07	1.48	1.24
R-FCN	7.32	6.16	5.39	5.21

Comparing these results to those in table 4.4, it is evident that even the EfficientNet-B0 PyTorch model achieves faster inference than the solution presented in [41]. However, it should be noted that in [67], a significantly higher number of objects needed to be detected, which could slow down inference due to the increased computational complexity required for processing multiple detections.
Finally, a specific vision-based algorithm for detecting and classifying small orbital debris using onboard optical cameras for near-real-time applications is analyzed in [70]. Their algorithm is for near real time applications and it was developed to address the challenges of in-situ small orbital debris detection. Table 4.7 presents how accuracy metrics vary depending on the FPS of image acquisition.
In particular, spots refer to all light sources detected in the video sequence, including stars and debris, while Objects are a subset of spots, that represents moving debris. From this results it is possible to notice that increasing the FPS (from 30 to 60) to

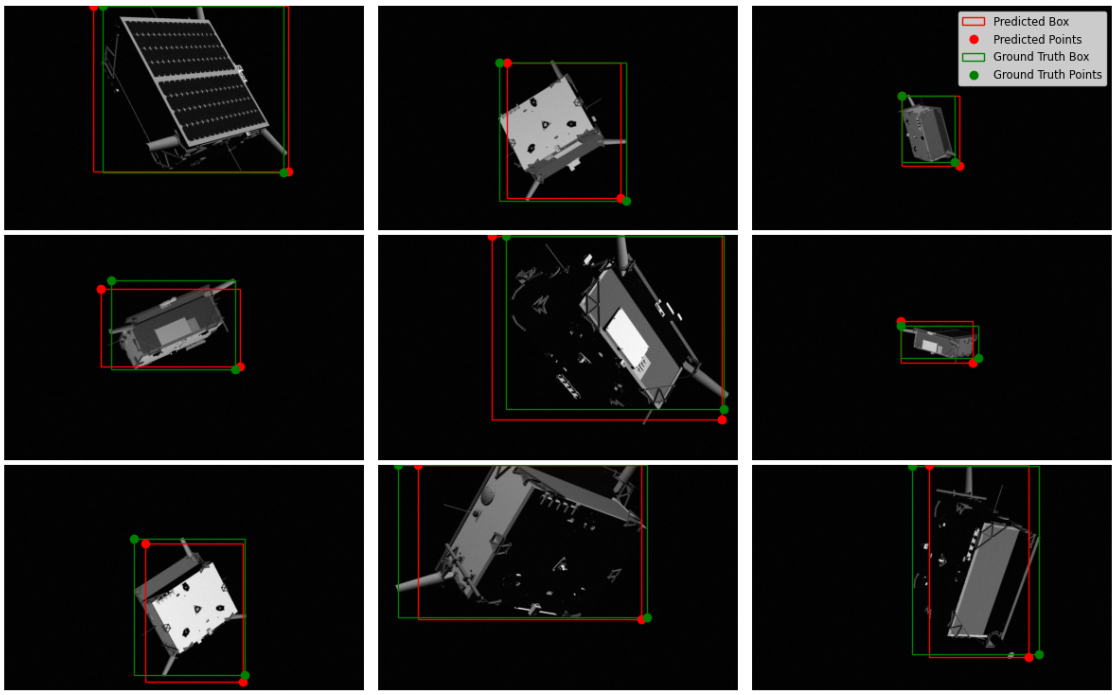
Table 4.7: Input Sample Data Specifications [70]

Sample Data	Frame Rate (FPS)	Duration (s)	Frame Count	Spots (Mean ± Std)	Objects (Mean ± Std)
Simple Foreground	30	6	180	134.7 ± 2.4	1.3 ± 0.7
Complex Foreground	30	30	900	117.7 ± 3.7	6.8 ± 2.0
Simple Background	60	60	3600	118.3 ± 2.1	5.3 ± 2.1
Complex Background	60	60	3600	114.0 ± 4.8	5.1 ± 2.3

improve object tracking over time and reduce tracking errors.
This analysis highlights the importance of frame acquisition frequency and its impact on real-time inference. Ensuring a high enough FPS is crucial for maintaining accurate tracking.
While the comparison may not be entirely fair, it helps illustrate the FPS range at which an operation can be considered real-time or near-real-time, particularly in the context of space debris tracking.

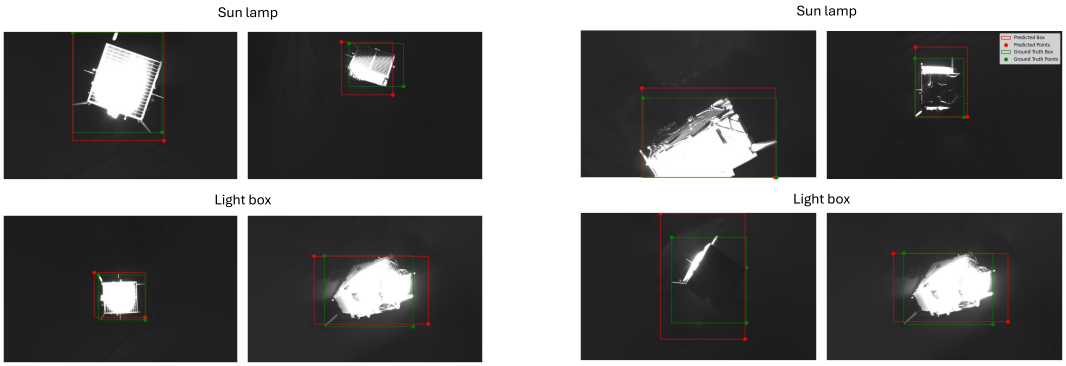


(a) YOLOv8n



(b) EfficientNet-B0

Figure 4.4: Visual comparison of YOLO and EfficientNet-B0 model predictions on 9 random test images



(a) EfficientNet-B0 (b) YOLOv8n

Figure 4.5: Visual comparison of YOLO and EfficientNet model predictions on HIL images

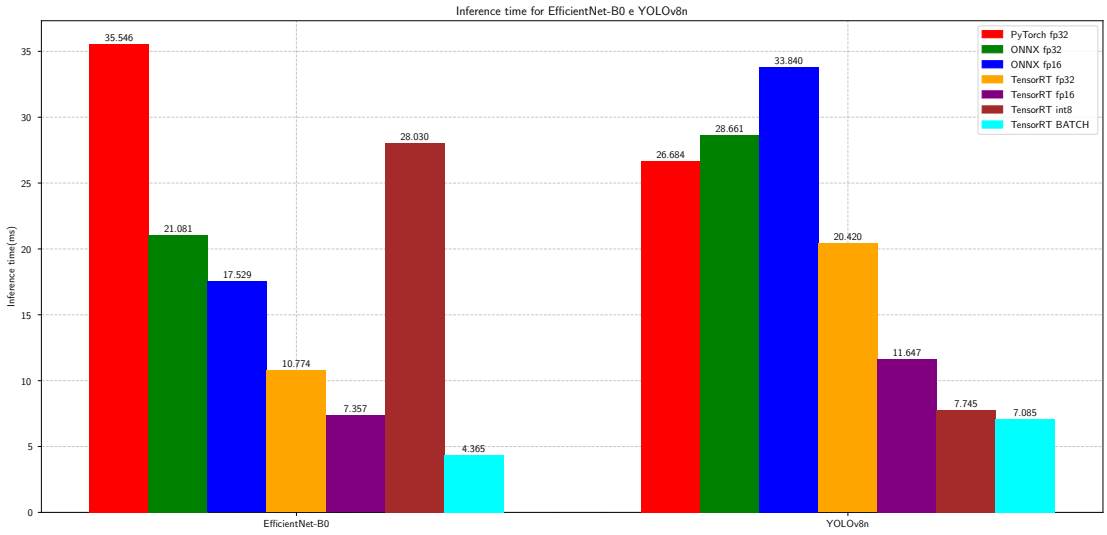


Figure 4.6: Inference time comparison between EfficientNet-B0 and YOLOv8n

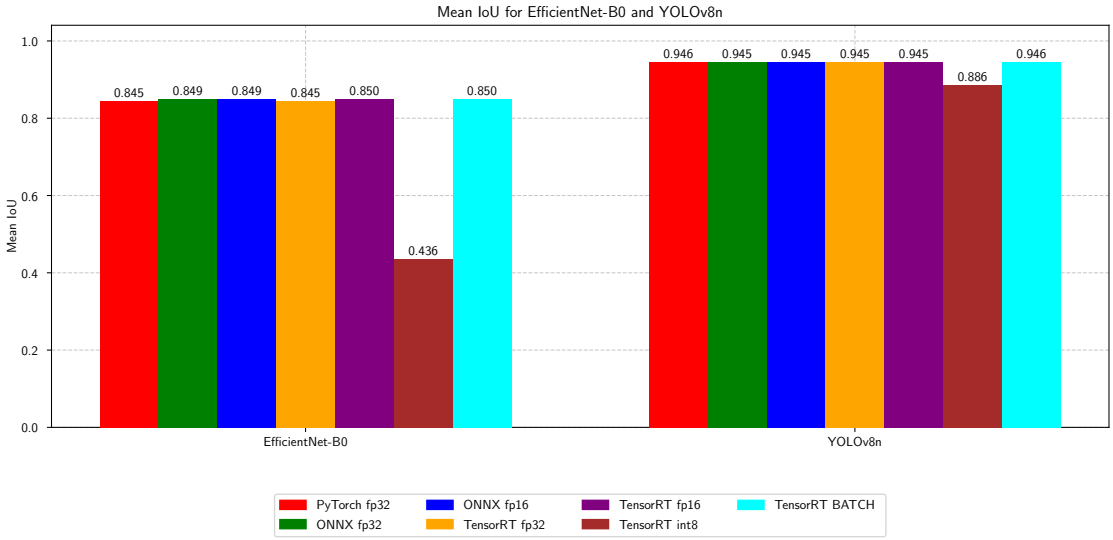


Figure 4.7: Accuracy comparison between EfficientNet-B0 and YOLOv8n

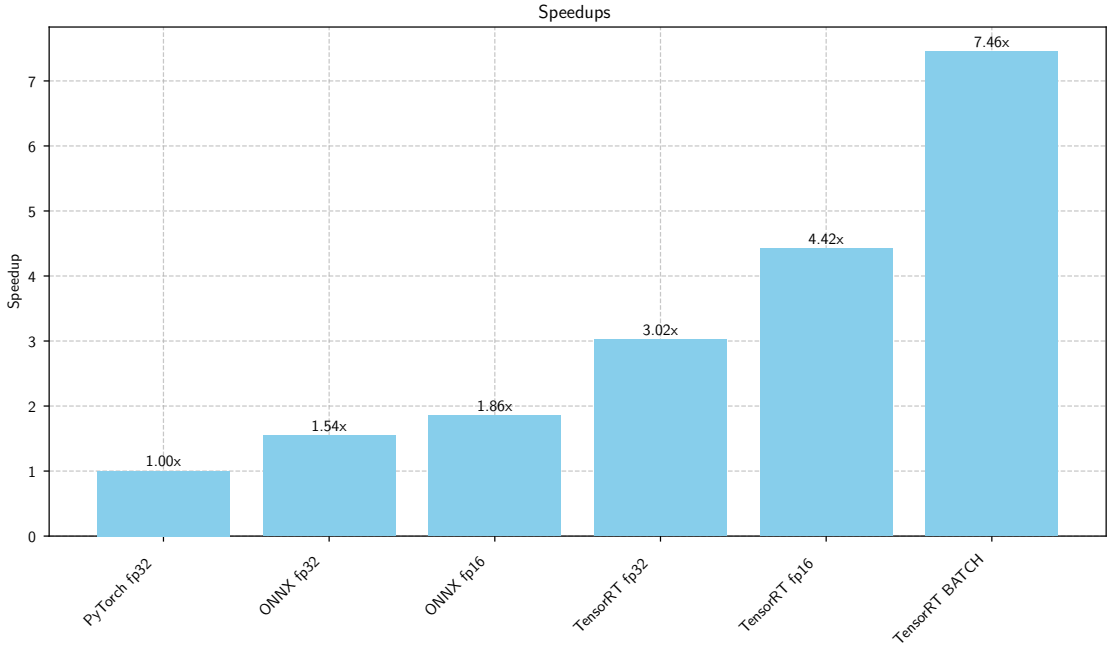


Figure 4.8: EfficientNet-B0 different model format inference speed speedups

Chapter 5

Conclusions

The main goal of this thesis was to develop a vision-based object detector for real-time space applications, specifically to assess the validity of this technology for future missions related to space debris management. To achieve this, the work primarily focused on optimizing inference speed, an essential requirement for autonomous spacecraft. In the context of space debris, high inference speed also contributes to improved tracking performance, particularly when detecting and following debris in star-populated image backgrounds, as confirmed by Zamani et al. [70].
To accomplish this goal, an EfficientNet-B0 model adapted for object detection was trained, along with a YOLOv8n model, on the SPEED+ dataset. During training for 100 epochs, the EfficientNet-B0 model achieved its best validation loss at epoch 89, with an RMSE of 4.94, reaching a maximum accuracy of 0.721 at epoch 90. The YOLOv8n model achieved a maximum accuracy of 0.994, with a Box Loss of 0.175, Class Loss of 0.098, and DFL of 0.856. From the training of both models, it is evident that YOLOv8n trained more efficiently, achieving higher accuracy.
This trend is confirmed during accuracy tests, where EfficientNet-B0 achieved a mean IoU over 1000 synthetic images of 0.849, compared to the 0.946 IoU of YOLOv8n. A significant loss in accuracy, as expected, was registered for both models during domain gap testing with 1000 HIL images, where an IoU of 0.382 and 0.442 was achieved, respectively, indicating a lack of robustness for real space scenarios.
During inference speed tests conducted on the Jetson Orin Nano, it was observed that model format and precision can significantly influence inference speed. In particular, when all model nodes are allocated to the chosen provider rather than being partitioned between the CPU and the provider, a notable speed improvement occurs. Specifically, the EfficientNet-B0 model initially had an inference time of 1841 ms when the PyTorch fp32 model was run on the CPU. This improved to the best inference speed of 7.35 ms and 136.05 FPS when using the ONNX model with the TensorRT provider at fp16 precision. This speed improvement did not affect mean accuracy, as the IoU increased slightly from 0.845 to 0.850. However, the quantization process carried out in this project did not prove to be an effective method for increasing inference speed due to discrepancies between model representations and the providers used.
Regarding YOLOv8n, the inference speed results revealed that the quantization

process was effective. The PyTorch model with fp32 precision running on the GPU achieved an inference speed of 26.684 ms and 37.47 FPS. This was further accelerated when using the .engine format on TensorRT, achieving an inference speed of 7.74 ms and 129.19 FPS with int8 precision. However, a relatively higher accuracy drop was registered, with the mean IoU decreasing from 0.946 to 0.886.
From these results, it is possible to state that the fastest EfficientNet-B0 model outperformed the YOLOv8n model in terms of speed, registering a 6.86 FPS higher score and a 0.39 ms faster inference at the cost of a 4.1% decrease in accuracy.
These results were then compared with significant but different literature studies, confirming that they fall within the range of real-time application requirements. In particular, the RemoveDEBRIS mission confirmed an inference time lower than 0.1 seconds [5], and the study on increasing inference speed for space applications conducted on the Jetson Nano by [41] recorded a best performance inference speed of 1.24 seconds with an SSD model.
Finally, another speed improvement was observed when inference was carried out on batches of images. For a batch size of 16 images, the EfficientNet-B0 ONNX model with the TensorRT provider in fp32 precision reached an inference speed of 4.36 ms and 229.09 FPS, while the fp32 YOLOv8n model in .engine format accelerated on TensorRT reached an inference speed of 7.085 ms and 141.1 FPS. Batch inference can significantly speed up the inference process, but it is not yet widely applied in real-time applications, as the results of the inference are only available once the entire batch has been processed, increasing latency.

5.1 Future works

While this thesis has demonstrated promising results, several areas warrant further investigation to enhance the robustness and real-world applicability of the proposed approach. First, addressing the domain gap remains a critical challenge. The current data augmentation techniques were not fully effective in mitigating this issue, suggesting that alternative strategies, such as unsupervised domain adaptation [14] and domain randomization [60], should be explored. Training on the entire dataset may also contribute to improved accuracy and allow for further hyperparameter optimization.
Additionally, different quantization techniques should be explored to fully benefit from the speed-up they offer. In particular, converting the EfficientNet-B0 ONNX model into a .engine file instead of using TensorRT providers could be investigated to ensure all model parameters are correctly converted. It is also necessary to ensure that the versions of TensorRT and CUDA installed on the device are compatible.
A crucial next step is the validation of the model in a real-world testbed environment. Instead of testing with a static printed image, the system should be deployed in a dedicated rendezvous simulation facility, typically featuring a dark room, two robotic arms for dynamic motion emulation, and a sun simulator. This setup will enable realistic evaluation under controlled lighting conditions and spacecraft-relative motion, ensuring the model's robustness for real chaser-target interactions.
Finally, other techniques described in this thesis, such as pruning or parallelization, may also be explored.

Bibliography

[1] Roya Afshar and Shuai Lu. "Classification and recognition of space debris and its pose estimation based on deep learning of CNNs". In: *HCI International 2020-Posters: 22nd International Conference, HCII 2020, Copenhagen, Denmark, July 19–24, 2020, Proceedings, Part I 22*. Springer. 2020, pp. 605–613.

[2] European Space Agency. *Hubble's impactful life alongside space debris*. ESA. 2020. URL: https://www.esa.int/Space_Safety/Hubble_s_impactful_life_alongside_space_debris.

[3] European Space Agency. *Space Debris FAQ: Frequently asked questions*. ESA. 2021. URL: https://www.esa.int/Space_Safety/Space_Debris/Space_Debris_FAQ_Frequently_asked_questions.

[4] Guglielmo S Aglietti et al. "RemoveDEBRIS: An in-orbit demonstration of technologies for the removal of space debris". In: *The Aeronautical Journal* 124.1271 (2020), pp. 1–23.

[5] Guglielmo S Aglietti et al. "The active space debris removal mission RemoveDebris. Part 2: In orbit operations". In: *Acta Astronautica* 168 (2020), pp. 310–322.

[6] Aliu Akinsemoyin et al. "Unmanned aerial systems and deep learning for safety and health activity monitoring on construction sites". In: *Sensors* 23.15 (2023), p. 6690.

[7] Syed Zahid Ali. *Principles of YoloV8*. Medium. 2023. URL: https://medium.com/@syedzahidali969/principles-of-yolov8-6a90564e16c3.

[8] Francesca Altieri et al. "Investigating the Oxia Planum subsurface with the ExoMars rover and drill". In: *Advances in Space Research* 71.11 (2023), pp. 4895–4903.

[9] Laith Alzubaidi et al. "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions". In: *Journal of big Data* 8 (2021), pp. 1–74.

[10] Alexander Amini et al. "Spatial uncertainty sampling for end-to-end control". In: *arXiv preprint arXiv:1805.04829* (2018).

[11] Kian Katanforoosh Andrew Ng and Younes Bensouda Mourri. *Neural Networks and DeepLearning*. Coursera. URL: https://www.coursera.org/learn/neural-networks-deep-learning/home/info.

[12] Jay E. Aronson. "Expert Systems". In: *Encyclopedia of Information Systems*. Ed. by Hossein Bidgoli. New York: Elsevier, 2003, pp. 277–289. ISBN: 978-0-12-227240-0. DOI: https://doi.org/10.1016/B0-12-227240-4/00067-8. URL: https://www.sciencedirect.com/science/article/pii/B0122272404000678.

[13] Herbert Bay. "Surf: Speeded up robust features". In: *Computer Vision—ECCV* (2006).

[14] Shai Ben-David et al. "Analysis of representations for domain adaptation". In: *Advances in neural information processing systems* 19 (2006).

[15] Saket Bhardwaj and Ajay Mittal. "A survey on various edge detector techniques". In: *Procedia Technology* 4 (2012), pp. 220–226.

[16] Gaudenz Boesch. *What is Intersection over Union (IoU)?* 2024. URL: https://viso.ai/computer-vision/intersection-over-union-iou/.

[17] Dr.-Ing. Benjamin Braun. *PRISMA Formation Flying Mission*. 2025. URL: https://www.dlr.de/en/rb/research-operation/research-projects/flight-dynamics-navigation-and-orbital-sustainability/gnss-technology-and-navigation/past-projects/prisma-formation-flying-misson.

[18] Ebubekir BUBER and Banu DIRI. "Performance Analysis and CPU vs GPU Comparison for Deep Learning". In: *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*. 2018, pp. 1–6. DOI: 10.1109/CEIT.2018.8751930.

[19] Alexander Buslaev et al. "Albumentations: Fast and Flexible Image Augmentations". In: *Information* 11.2 (2020), p. 125. DOI: 10.3390/info11020125. URL: https://www.mdpi.com/2078-2489/11/2/125.

[20] Hongrong Cheng, Miao Zhang, and Javen Qinfeng Shi. "A Survey on Deep Neural Network Pruning: Taxonomy, Comparison, Analysis, and Recommendations". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 46.12 (2024), pp. 10558–10578. DOI: 10.1109/TPAMI.2024.3447085.

[21] Dwith Chenna. "Quantization of Convolutional Neural Networks: A Practical Approach". In: *International Journal for Research Trends and Innovation* (2023).

[22] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[23] NVIDIA DEVELOPER. *JetPack SDK*. https://developer.nvidia.com/embedded/jetpack.

[24] NVIDIA DEVELOPER. *NVIDIA TensorRT*. https://developer.nvidia.com/tensorrt/.

[25] ONNX Runtime developers. *ONNX Runtime*. https://onnxruntime.ai/. Version: x.y.z. 2021.

[26] P Kingma Diederik. "Adam: A method for stochastic optimization". In: *(No Title)* (2014).

[27] Tausif Diwan, G Anirudh, and Jitendra V Tembhurne. "Object detection using YOLO: Challenges, architectural successors, datasets and applications". In: *multimedia Tools and Applications* 82.6 (2023), pp. 9243–9275.

[28] Clément Ernould et al. "Chapter One - Measuring elastic strains and orientation gradients by scanning electron microscopy: Conventional and emerging methods". In: ed. by Martin Hÿtch and Peter W. Hawkes. Vol. 223. Advances in Imaging and Electron Physics. Elsevier, 2022, pp. 1–47. DOI: https://doi.org/10.1016/bs.aiep.2022.07.001. URL: https://www.sciencedirect.com/science/article/pii/S1076567022000544.

[29] Jason L Forshaw et al. "RemoveDEBRIS: An in-orbit active debris removal demonstration mission". In: *Acta Astronautica* 127 (2016), pp. 448–463.

[30] Stanislav Ganea. "Deep Learning for Spacecraft Detection and Classification in Orbital Operations". Master thesis. Blogna: Alma Mater Studiorum - University of Bologna, 2024.

[31] Kaifeng Gao et al. "Julia language in machine learning: Algorithms, applications, and open issues". In: *Computer Science Review* 37 (2020), p. 100254.

[32] Hossein Gholamalinezhad and Hossein Khosravi. "Pooling methods in deep neural networks, a review". In: *arXiv preprint arXiv:2009.07485* (2020).

[33] Justin Goodwill, Christopher Wilson, and James MacKinnon. "Current AI technology in space". In: *Precision Medicine for Long and Safe Permanence of Humans in Space*. Elsevier, 2025, pp. 239–250.

[34] Anna Heiney. *Demo-1: Watch Crew Dragon Hatch Closure*. NASA. 2019. URL: https://blogs.nasa.gov/commercialcrew/2019/03/page/2/.

[35] *Jetson Modules*. NVIDIA. 2025. URL: https://developer.nvidia.com/embedded/jetson-modules.

[36] Ari Jonsson, Robert A. Morris, and Liam Pedersen. "Autonomy in Space: Current Capabilities and Future Challenge". In: *AI Magazine* 28.4 (Dec. 2007), p. 27. DOI: 10.1609/aimag.v28i4.2066. URL: https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/2066.

[37] Matthew Kaufman. *NASA Trains Machine Learning Algorithm for Mars Sample Analysis*. 2025. URL: https://www.nasa.gov/solar-system/nasa-trains-machine-learning-algorithm-for-mars-sample-analysis.

[38] Donald J Kessler and Burton G Cour-Palais. "Collision frequency of artificial satellites: The creation of a debris belt". In: *Journal of Geophysical Research: Space Physics* 83.A6 (1978), pp. 2637–2646.

[39] Robin Larsson Nordström et al. "Flight results from SSCS GNC experiments within the PRISMA formation flying mission". In: *61st International Astronautical Congress 2010, IAC 2010* 7 (Jan. 2010), pp. 6032–6041.

[40] Xiang Li et al. "Generalized focal loss: Learning qualified and distributed bounding boxes for dense object detection". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 21002–21012.

[41] Martina Lofqvist and José Cano. "Accelerating deep learning applications in space". In: *arXiv preprint arXiv:2007.11089* (2020).

[42] David G Lowe. "Object recognition from local scale-invariant features". In: *Proceedings of the seventh IEEE international conference on computer vision*. Vol. 2. Ieee. 1999, pp. 1150–1157.

[43] Christine Mwase et al. "Communication-efficient distributed AI strategies for the IoT edge". In: *Future Generation Computer Systems* 131 (2022), pp. 292–308.

[44] Niall O'Mahony et al. "Deep learning vs. traditional computer vision". In: *Advances in Computer Vision: Proceedings of the 2019 Computer Vision Conference (CVC), Volume 1 1*. Springer. 2020, pp. 128–144.

[45] Rafael Padilla, Sergio L. Netto, and Eduardo A. B. da Silva. "A Survey on Performance Metrics for Object-Detection Algorithms". In: *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*. 2020, pp. 237–242. DOI: 10.1109/IWSSIP48289.2020.9145130.

[46] Tae Ha Park et al. "SPEED+: Next Generation Dataset for Spacecraft Pose Estimation across Domain Gap". In: *CoRR* abs/2110.03101 (2021). arXiv: 2110.03101. URL: https://arxiv.org/abs/2110.03101.

[47] Nancy J. Pekar. *Fresh Eyes on Mars: Mars 2020 Lander Vision System Tested through NASA's Flight Opportunities Program*. NASA. 2016. URL: https://www.nasa.gov/missions/mars-2020-perseverance/fresh-eyes-on-mars-mars-2020-lander-vision-system-tested-through-nasas-flight-opportunities-program/.

[48] Swathi Pothuganti. "Review on over-fitting and under-fitting problems in Machine Learning and solutions". In: *Int. J. Adv. Res. Electr. Electron. Instrum. Eng* 7.9 (2018), pp. 3692–3695.

[49] *PyTorch*. 2025. URL: https://pytorch.org/.

[50] Juan F Rodríguez et al. "Use of neural networks for tsunami maximum height and arrival time predictions". In: *GeoHazards* 3.2 (2022), pp. 323–344.

[51] Ivan Rodriguez-Ferrandez et al. "Exploring Total Ionizing Dose Radiation Effects Across Generations of NVIDIA Jetson Devices: A Comparative Analysis". In: *2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE. 2024, pp. 1–6.

[52] Mark Sandler et al. "Mobilenetv2: Inverted residuals and linear bottlenecks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[53] Iqbal H Sarker. "Deep learning: a comprehensive overview on techniques, taxonomy, applications and research directions". In: *SN computer science* 2.6 (2021), p. 420.

[54] Minghe Shan, Jian Guo, and Eberhard Gill. "Review and comparison of active space debris capturing and removal methods". In: *Progress in aerospace sciences* 80 (2016), pp. 18–32.

[55] Himadri Sharma. *What Is Deep Learning and How to Use It in Marketing*. mailmodo. 2025. URL: https://www.mailmodo.com/guides/deep-learning-marketing/.

[56] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. "Activation functions in neural networks". In: *Towards Data Sci* 6.12 (2017), pp. 310–316.

[57] K Sindhu Meena and S Suriya. "A survey on supervised and unsupervised learning techniques". In: *Proceedings of international conference on artificial intelligence, smart grid and smart city applications: AISGSC 2019*. Springer. 2020, pp. 627–644.

[58] Mingxing Tan and Quoc Le. "Efficientnet: Rethinking model scaling for convolutional neural networks". In: *International conference on machine learning*. PMLR. 2019, pp. 6105–6114.

[59] Juan Terven, Diana-Margarita Córdova-Esparza, and Julio-Alejandro Romero-González. "A comprehensive review of yolo architectures in computer vision: From yolov1 to yolov8 and yolo-nas". In: *Machine learning and knowledge extraction* 5.4 (2023), pp. 1680–1716.

[60] Josh Tobin et al. "Domain randomization for transferring deep neural networks from simulation to the real world". In: *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 2017, pp. 23–30.

[61] *Trends*. Papers with code. 2025. URL: https://paperswithcode.com/trends.

[62] *Ultralytics Documentation*. Ultralytics. 2025. URL: https://docs.ultralytics.com/.

[63] Qi Wang et al. "A comprehensive survey of loss functions in machine learning". In: *Annals of Data Science* (2020), pp. 1–26.

[64] Yanzhi Wang et al. "Non-structured dnn weight pruning considered harmful". In: *arXiv preprint arXiv:1907.02124* 2 (2019).

[65] Olivia Weng. "Neural network quantization for efficient inference: A survey". In: *arXiv preprint arXiv:2112.06126* (2021).

[66] Qiang Wu and Kenneth R. Castleman. "Chapter Seven - Image Segmentation". In: *Microscope Image Processing (Second Edition)*. Ed. by Fatima A. Merchant and Kenneth R. Castleman. Second Edition. Academic Press, 2023, pp. 119–152. ISBN: 978-0-12-821049-9. DOI: https://doi.org/10.1016/B978-0-12-821049-9.00003-4. URL: https://www.sciencedirect.com/science/article/pii/B9780128210499000034.

[67] Gui-Song Xia et al. "DOTA: A large-scale dataset for object detection in aerial images". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 3974–3983.

[68] Weizheng Xu, Youtao Zhang, and Xulong Tang. "Parallelizing DNN training on GPUs: Challenges and opportunities". In: *Companion Proceedings of the Web Conference 2021*. 2021, pp. 174–178.

[69] Raniah Zaheer and Humera Shaziya. "A study of the optimization algorithms in deep learning". In: *2019 third international conference on inventive systems and control (ICISC)*. IEEE. 2019, pp. 536–539.

[70] Yasin Zamani et al. "A robust vision-based algorithm for detecting and classifying small orbital debris using on-board optical cameras". In: *Advanced Maui Optical and Space Surveillance Technologies Conference*. M19-7620. 2019.

[71] Wei Zhou et al. "EARDS: EfficientNet and attention-based residual depthwise separable convolution for joint OD and OC segmentation". In: *Frontiers in Neuroscience* 17 (2023), p. 1139181.