ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Corso di Laurea in:

Ingegneria Aerospaziale

Classe L-9

Elaborato finale di laurea in Satelliti e Missioni Spaziali

Deep Learning for Spacecraft Detection and Classification in Orbital Operations

Candidato:
Stanislav Ganea

Relatore:
Prof. Dario Modenini

Correlatore:
Ing. Roman Prokazov

A.A. 2023-2024

Abstract

As humanity's in-orbit activities increased, so did the number of debris. The same can't be said about policies that should have regulated such activities, especially in the first decades of space conquest. Today, after many years of unregulated reign, we face all the problems that this phenomenon brings. The rapid advancement of space exploration has driven the need for automated systems capable of accurately identifying and classifying spacecraft under various conditions. This thesis presents a machine learning-based approach to spacecraft detection and classification using image data. A two-stage process is implemented on the NASA PoseBowl dataset: first, YOLO's object detection model is implemented to localize spacecraft within the image frames. Next, MobileNetV3, a classification model fine-tuned to detected spacecraft, leveraging cropped images to reduce background interference and improve classification accuracy. The classification model is trained on images from a chaser spacecraft's perspective, achieving a high level of accuracy in both training and testing accuracy after extensive model tuning and refinement. This thesis work demonstrates that combining object detection with image classification significantly enhances the accuracy of spacecraft identification, offering a robust solution for future space operations such as docking, rendezvous, and other on-orbiting servicing operations. This research contributes to the growing field of autonomous spacecraft systems, with potential applications in satellite management, more in particular in space debris removal–a rising concern that needs to be addressed for a sustainable future.

Abstract (Italian Version)

Con l'aumento delle attività in orbita, è cresciuto anche il numero di detriti spaziali. Lo stesso non si può dire delle politiche volte a regolamentare tali attività, soprattutto nei primi decenni della conquista dello spazio. Oggi, dopo molti anni di gestione non regolamentata, ci troviamo a fronteggiare tutte le problematiche derivanti da questo fenomeno. Il rapido progresso dell'esplorazione spaziale ha reso indispensabile lo sviluppo di sistemi automatizzati in grado di identificare e classificare accuratamente i veicoli spaziali in condizioni variabili. Questa tesi presenta un approccio basato sul machine learning per il rilevamento e la classificazione tramite immagini di veicoli spaziali. È stato implementato un processo in due fasi sul dataset PoseBowl della NASA: in primo luogo, è stato utilizzato il modello di rilevamento oggetti YOLO per localizzare i veicoli spaziali nei fotogrammi; successivamente, è stato impiegato MobileNetV3, un modello di classificazione ottimizzato per rilevare gli spacecraft, sfruttando immagini ritagliate per ridurre l'interferenza dello sfondo e migliorare l'accuratezza della classificazione. Il modello di classificazione è stato addestrato su immagini riprese dal punto di vista di un veicolo spaziale inseguitore, raggiungendo un'elevata accuratezza sia in fase di addestramento sia di test grazie a un'attenta ottimizzazione e calibrazione del modello. Questo lavoro di tesi dimostra che combinare il rilevamento di oggetti con la classificazione delle immagini migliora significativamente l'accuratezza nell'identificazione degli spacecraft, offrendo una soluzione robusta per future operazioni spaziali come attracco, rendezvous e altre attività di servizio in orbita. Questa ricerca contribuisce al crescente campo dei sistemi autonomi per veicoli spaziali, con potenziali applicazioni nella gestione dei satelliti e, in particolare, nella rimozione dei detriti spaziali, una questione emergente che necessita una soluzione per garantire un futuro sostenibile.

Contents

List of Figures

Chapter 1

Introduction

Space debris is a growing concern as Earth's orbit becomes increasingly cluttered with defunct satellites, spent rocket stages, and fragments from collisions. This debris creates significant risks to operational satellites, space missions, and the International Space Station (ISS). Even small pieces of debris, traveling at high velocities, can cause catastrophic damage upon impact as shown in Figure 1.1. The accumulation of space debris not only threatens the safety of current and future space operations but also increases the likelihood of cascading collisions, known as the Kessler Syndrome [4], which could render parts of Earth's orbit unusable for generations. Addressing this issue requires global cooperation and innovative solutions to remove or mitigate debris, that is why in the last years there has been a positive trend of researching the topic with many remarkable results like the first future Active Debris Removal/In Orbit Servicing (ADRIOS) mission assigned to ClearSpace [3] by the European Space Agency (ESA). Figure 1.2 shows ClearSpace spacecraft to perform deorbiting.

As the number of satellites in orbit increases drastically each year, so does the risks. These problems combined with the soon-not-sustainable efforts of humans and conventional algorithms to tackle all spacecraft operations make a new necessity arise, automation in spacecraft operations, particularly vision-based tasks. This is where Machine Learning (ML) comes into play. It is a well-established technology on Earth that still has to find its way in the space domain. The objective of this thesis is to demonstrate the efficiency of such techniques, how they can be implemented, and why they seem to pose an advantage and a necessity for futures generations of spacecraft.

A traditional approach to mitigate these issues is collision avoidance systems. It represent a critical technological asset in managing the increasingly congested orbital environment. At their core, these systems are designed to predict and prevent potential impacts between spacecraft and the vast array of orbital debris circling the Earth. The fundamental challenge lies in processing complex trajectorial data in real-time, making split-second decisions that can mean the difference between mission success and catastrophic failure. Traditional collision avoidance approaches have relied on ground-based tracking networks and predictive mathematical models, for example the General Method for Calculating Satellite Collision Probability [8]. The primary system, known as the Space Surveillance Network (SSN), maintained by the United States Space Force, tracks approximately 27,000 pieces of orbital debris larger than 10 centimeters. However, this system faces significant limitations. The sheer number of smaller debris fragments is estimated at over 170 million pieces smaller than 10 centimeters. Modern spacecraft, particularly those in low Earth orbit (LEO), are equipped with increasingly sophisticated collision avoidance mechanisms. The ISS, for instance, maintains a robust collision avoidance protocol that involves multiple layers of protection. When a potential collision is detected, the station can perform debris avoidance maneuvers, using its thrusters to alter its orbital trajectory. These maneuvers are typically planned days in advance, based on precise orbital calculations and debris tracking data.

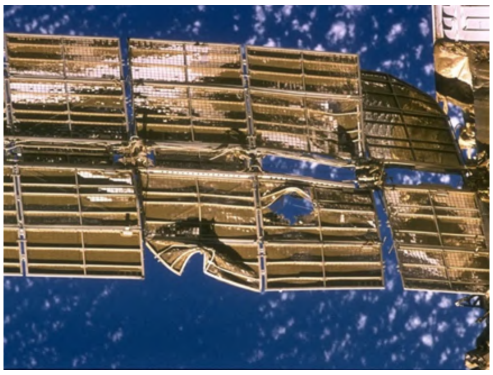


Figure 1.1: Space Debris Damage. *Image credit:* ESA *NASA*

Figure 1.2: ClearSpace spacecraft deorbiting a satellite. *Image credit*

The work will be structured as follows. The principles that govern machine learning–more in particularly–deep learning will be explained in Chapter 2. Chapter 3 will expand on the framework used for this work, and which state-of-the-art algorithms have been chosen to perform training. We will then move onto showing how data is processed and prepared to be fed to the model in Chapter 4. Chapter 5 will present the final results, from showing the full training process of both models to highlighting the difference that object detection makes in classification accuracy. Moreover, inspired by the work of the OpenAI team [2] on enhancing AI interpretability, it will be offered a glimpse into the inner workings of a neural network (NN) as it processes an image, in the end we will visualize the network's multi-dimensional output space, showcasing how each class is represented through both 2D embedding spaces and interactive 3D projections, along with its associated manifold. Finally in Chapter 6 all conclusions are drawn, it is discussed how all the work could be improved, and what real-world applications could benefit from this technology.

Chapter 2

Deep Learning

2.1 Introduction to Machine Learning

Before introducing deep learning, it's important to provide an overview of machine learning (ML). Machine learning is a subfield of computer science that has been around for several decades, but its popularity has increased in the last years being a computational power-enabled technology. At its core, ML aims to enable systems to perform tasks autonomously without the need for explicitly programming every possible scenario. Essentially, these systems are designed to recognize patterns in data. When these patterns are simple, predictions are relatively straightforward; however, as the complexity of the patterns increases, so does the difficulty of making accurate predictions.

In more detail, all problems can be framed as a function approximation task, where the goal of a ML model is to predict this function as accurately as possible. The model achieves this by learning from the input data it is provided. Generally, the more data that is available, the easier it becomes for the model to identify patterns and improve its predictions. Historically, the limited availability of data, alongside insufficient computational power, were major obstacles to the adoption of such technology.

2.2 Principles of Deep Learning

Deep Learning specifically relies on Artificial Neural Networks (ANNs). The core idea is to connect many artificial neurons, each retaining some type of information, and make them work together to perform specific tasks. Conveniently, the type of information processed by these neurons is numerical, and the tasks involve mathematical operations. The goal is to produce some output given a certain input.

The first ANN, the Perceptron, a single layer network, was created by Frank Rosenblatt in 1957 [10]. It was a simple binary classifier. Over the following decades, advancements were made, leading to the understanding that neurons and their connections can be arranged in various ways, achieving different layouts that perform better for specific tasks. One of the most notable architectures is the Convolutional Neural Network (CNN), which is particularly effective for image classification. In 1989, a CNN known as LeNet-1 [5] successfully classified a dataset of handwritten digits. The fact that we still use similar architectures today highlights that the algorithms were correct, although computationally too costly for the time to achieve something meaningful. Some of these architectures are illustrated in Figure 2.1.

There are many approaches to enable a model to learn (e.g., unsupervised learning, reinforcement learning, etc.). The problem we are addressing falls under supervised learning, where the model is trained on a labeled dataset, meaning the input data is paired with the correct output (target). In our case, each image containing a spacecraft is labeled with the appropriate class. The goal is to learn a mapping from inputs to outputs, allowing the model to make predictions on new, unseen data.

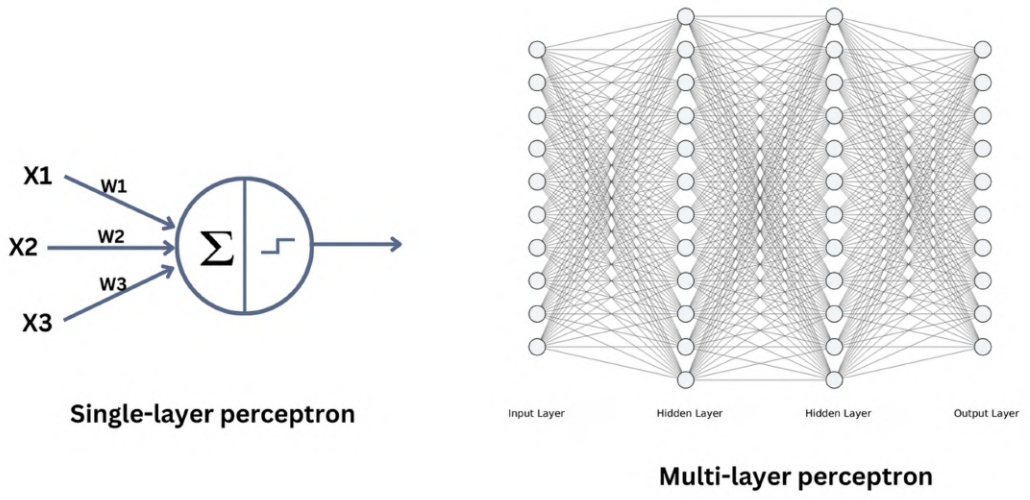


Figure 2.1: In the single-layer perceptron we can see inputs (x), weights of each connection (w), a weighted sum and a step activation function. By only observing the symbolic architecture we can appreciate the rise in complexity in a multi-layer design. *Image credit*

In deep learning, optimizers and loss functions play fundamental roles in the training process.

An **optimizer** is an algorithm that updates the parameters of a neural network (weights and biases) to minimize the error calculated by the loss function. It does this by leveraging the gradients of the loss with respect to the parameters, computed during backpropagation. Different optimizers follow varying strategies for updating the parameters:
Stochastic Gradient Descent (SGD): A foundational optimizer that updates the parameters using the gradients of a randomly selected batch of data. Despite its simplicity, SGD often performs well, particularly with proper fine-tuning of hyperparameters like the learning rate.
Adam: Combines momentum (which accelerates learning) and adaptive learning rates (which adjust step sizes for each parameter), often making it a strong choice for many problems.
RMSProp: An optimizer that adapts the learning rate for each parameter by normalizing gradients, making it useful for problems with non-stationary objectives.

The **learning rate** ("lr") is a crucial hyperparameter that determines the step size at each iteration when updating the parameters. If the learning rate is too large, the optimization process may overshoot the optimal solution, leading to instability. Conversely, if it is too small, convergence may become too slow.

The **loss function** is a metric that quantifies how well the model's predictions align with the true labels. It guides the optimizer by providing feedback on the quality of predictions.

2.3 Convulutional Neural Networks For Image Classification

A convolutional neural network (CNN) is a type of deep neural network composed of several interconnected layers, which will be explained in detail in this section. Many of the theoretical concepts discussed here are will be illustrated specifically for our dataset in Chapter 5, with an initial schematic of a CNN shown in Figure 2.2.

To begin, it is important to understand how the input image will be processed by the computer. An image is best represented as a matrix, with the numbers of rows and columns corresponding to the image's height and width, and each pixel in the matrix assigned a value based on its intensity, typically ranging from 0 to 255. In grayscale images, 0 represents black, 255 represents white, and intermediate values represent shades of gray. For color images, three matrices (one each for Red, Green, and Blue channels) are stacked together, as any color can be formed by mixing varying intensities of these three colors.

With the input representation defined, the CNN architecture can be summarized as follows. The first operation involves convolution, which takes another matrix (usually orders of magnitude smaller) called a filter, kernel, or feature detector, and sliding it over the input matrix. A convolution is mathematically the dot product of two functions as shown in Equation (2.1), and in this case, it generates a new function/matrix — the feature map. The first kernels are responsible to detect specific shallow features, such as edges, corners, or simple patterns. A single convolutional layer typically contains multiple kernels (e.g., 16 in our case), each generating its own feature map.

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau \tag{2.1}$$

Following the convolutional layers, we use pooling layers, which downsample the feature maps, keeping only the most significant features and reducing the spatial dimensions of the data. This not only speeds up computation but also improves the network's generalization capabilities, reducing the risk of *overfitting*. By stacking multiple convolutional and pooling layers, the network can detect increasingly complex features, allowing it to capture objects, shapes, and intricate patterns as it progresses, ultimately yielding a final, small matrix representation of the input. This entire process of convolution and pooling is known as feature extraction.

Once features have been extracted, the network needs to classify them. This is done through fully connected layers, similar to those in a multilayer perceptron (see Figure 2.1). In our model, the classifier head is composed of 2048 neurons, which will be trained by adjusting the weights of each connection during training through backpropagation, a widely used algorithm that iteratively reduces the network's error by propagating it backward through the layers and updating weights using gradient descent. The update rule for gradient descent is shown in Equation (2.2), where θ represents the parameters being optimized, t is the current iteration, η is the learning rate and $J(\theta)$ is the cost function with respect to θ.

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta) \tag{2.2}$$

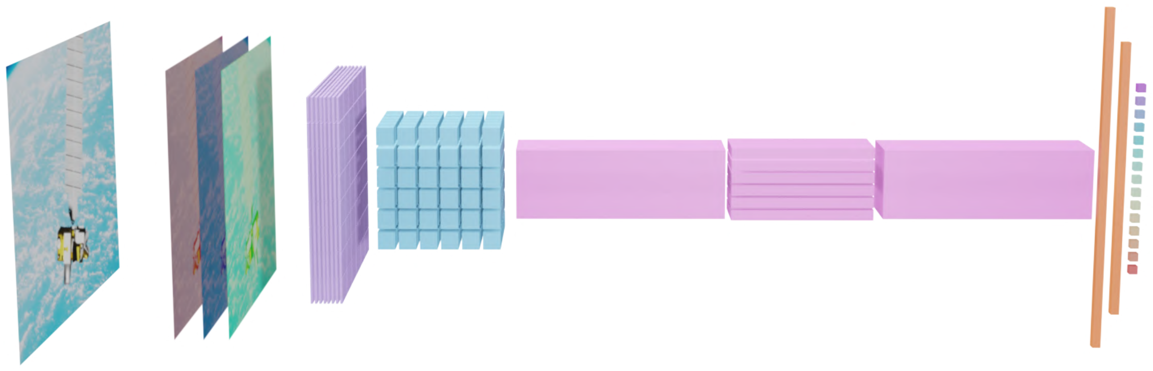


Figure 2.2: Generic CNN architecture. The blocks represent convolution and pooling until the fully connected layers are reached, and finally we have the last 15 neurons representing each class.

Chapter 3

Methods And Tools

3.1 The PyTorch Framework

PyTorch is an open-source deep learning framework, widely used for constructing and training neural networks. Built on a foundation of dynamic computation graphs, PyTorch provides flexibility in model development, making it particularly suitable for research and iterative experimentation. It supports tensor computation on both CPU and GPU, enabling efficient handling of large-scale data. Additionally, PyTorch offers a robust library of pre-built neural network layers and a wide range of tools for model optimization, making it a common choice for developing and deploying machine learning and deep learning applications.

3.2 Visual Studio Code & Google Colab

Visual Studio Code (VSCode) is an open-source code editor that is widely used for its versatility and powerful features. It supports a wide range of programming languages and offers a variety of extensions, making it an ideal environment for coding tasks. For machine learning projects, such as image classification, VSCode provides features like integrated debugging, version control, and support for Jupyter notebooks, which are very useful for development and testing.

Google Colab is a cloud-based platform that provides an accessible environment for running Python code, it is particularly useful in ML given that one of its most significant advantages is the provision of free access to graphical processing units (GPUs) and tensor processing units (TPUs) developed specifically for ML training, which are essential for accelerating the training of deep learning models. This makes Google Colab an invaluable tool for ML-related projects, where large datasets and complex neural networks can require substantial computational power.

3.3 State Of The Art Algorithms

3.3.1 MobileNet

MobileNetV3 [11] was chosen as the backbone architecture for the classification network for this study due to its efficient design and strong performance on resource-constrained systems. The network, developed by Google researchers in 2018, is an improvement over the original MobileNet architecture, featuring an innovative design that uses depthwise separable convolutions and a unique inverted residual structure.

These architectural choices allow the model to first expand the number of channels for feature extraction, then filter features through efficient depthwise convolutions, and finally project them back to a lower-dimensional representation. This approach significantly reduces computational complexity and model size compared to traditional convolutional neural networks, while maintaining high accuracy levels. In our implementation, MobileNetV2 proved to be particularly suitable due to its balance of performance and efficiency, processing our dataset with considerably fewer parameters and computational resources than would be required by standard architectures.

3.3.2 You Only Look Once

YOLOv8, released by Ultralytics in 2023 [9], was selected as the main object detection network (ODN) in this research because it is a state-of-the-art object detection network which is well-suited for real time applications. The model works by analyzing images just once to detect multiple objects simultaneously, unlike older methods that needed to scan images multiple times. It uses a special backbone system called CSP-Darknet53 [1] to extract important features from images, and a feature fusion network (PANet) that helps the model better understand objects at different sizes and distances. One of the key improvements in YOLOv8 is its anchor-free approach, which means it can detect objects more naturally without predefined box sizes. The model also includes modern training techniques that help it learn more effectively and make more accurate predictions. In our implementation, YOLOv8 proved particularly valuable because it could process our images with high inference speed while maintaining sufficient accuracy. Figure 3.1 shows an overlay of the model components.

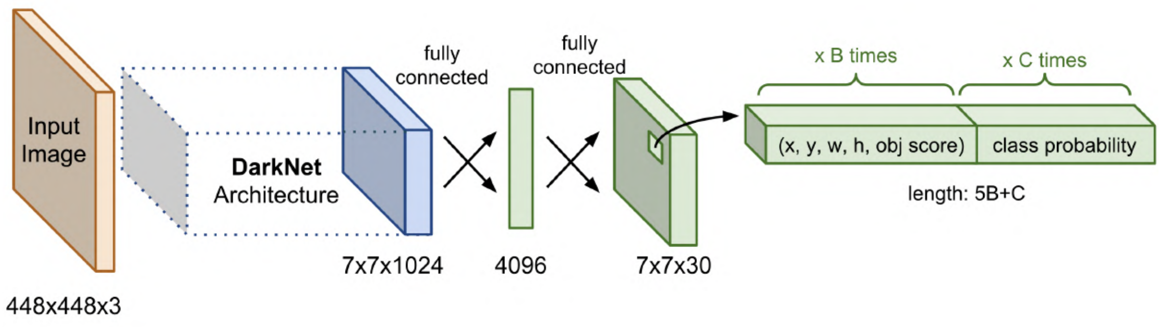


Figure 3.1: The network architecture of YOLO. *Image credit*

Chapter 4

The Dataset

The dataset used for the official Pose Bowl Detection for Spacecraft Detection and Pose Estimation Challenge issued by NASA was chosen, as it's one of the few and newest datasets of spacecraft imagery available alongside SPEED+ [7] and SPARK [6].

4.1 Dataset Description

The dataset consists of 8021 images taken randomly out of the full dataset. It contains simulated images of spacecraft taken from a nearby location in space, as if from the perspective of a chaser spacecraft. Images were created using the 3D software Blender using models of representative host spacecraft against simulated backgrounds. A limited number of models and backgrounds were used to generate images; models and backgrounds appear in multiple images. Distortions were applied to some images in post-processing to realistically simulate image imperfections that result from camera defects or field conditions including blur, hot pixels and random noise. An example of images from the dataset can be found in Figure 4.1. Moreover the dataset is composed of 15 different classes each representing a different spacecraft, the exact number of samples for each class can be found in Table 4.1. In addition to the images ground truth data is available in csv file format. This data will later be crucial in the training process both for classification and object detection as it contains labelled images and spacecrafts bounding box coordinates respectively.

Classes	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Samples	430	457	447	510	532	373	520	526	549	575	510	667	653	654	618

Table 4.1: Distribution of samples across classes.

4.2 Dataset Preparation

4.2.1 Data Augmentation

It is assesed how deep learning can be prone to the phenomenon of overfitting, in particular, computer vision is very susceptible to it. There are some strategies to prevent the model from learning too specific features of the training dataset and help it generalize as well as possible, perhaps one of these techniques is data augmentation.

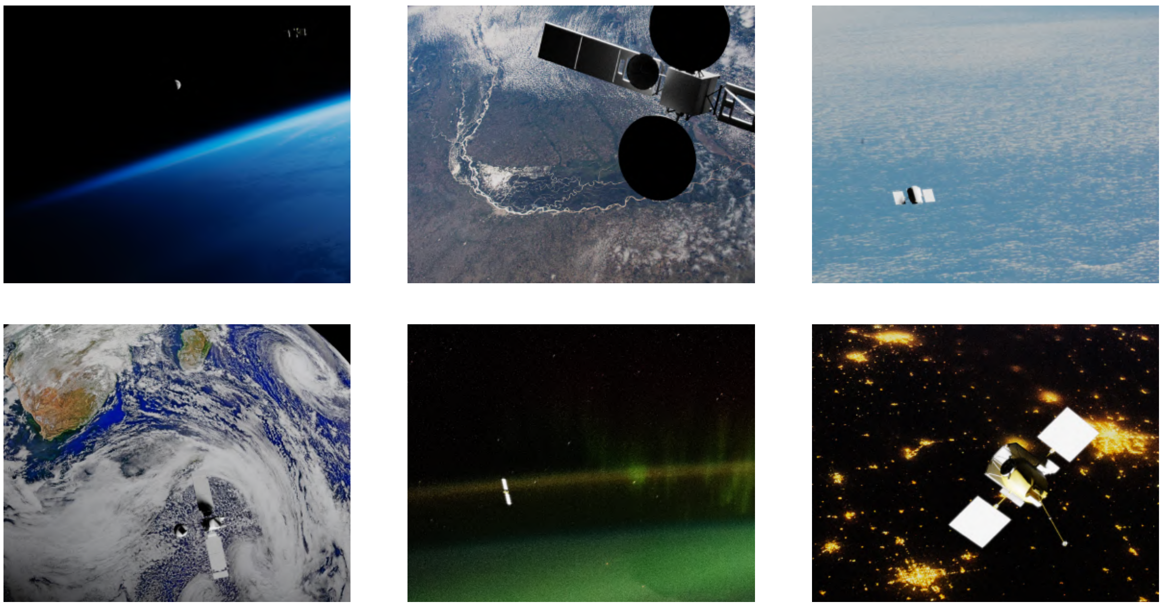


Figure 4.1: Examples of spacecraft images from PoseBowl dataset.

In particular, if the goal is to make the model see the image a little unclear, to not learn some of those very deep and too specific patterns, then, the easiest way to do so is to actually artificially worsen the quality of said images (e.g. random crops, noise, blur, rotations, hot pixels etc...). This process increases the diversity of the dataset without the need for additional data collection, helping simulate a broader range of scenarios and variations that the model might encounter in real-world applications, achieving overall higher robustness. Now when the model is shown new unseen test images, thanks to the generalization capabilities it is more likely to predict accurately. An example of data augmentation is shown in Figure 4.2.

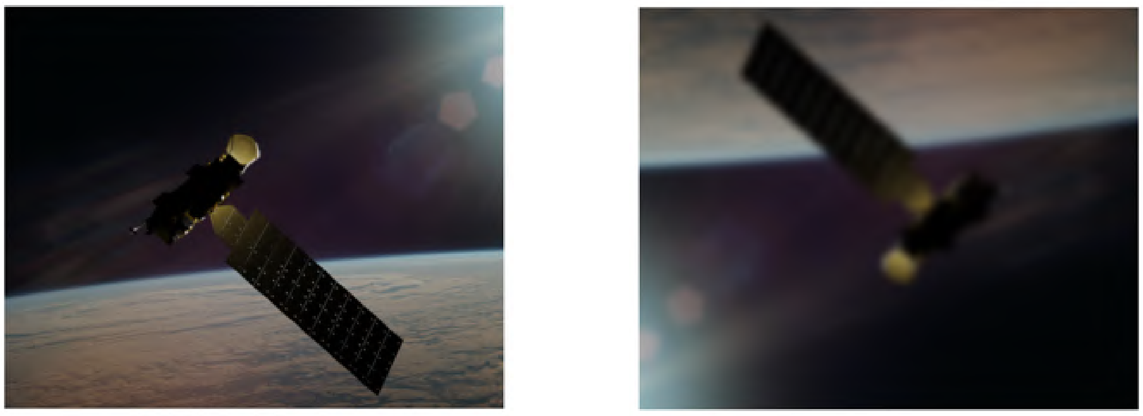
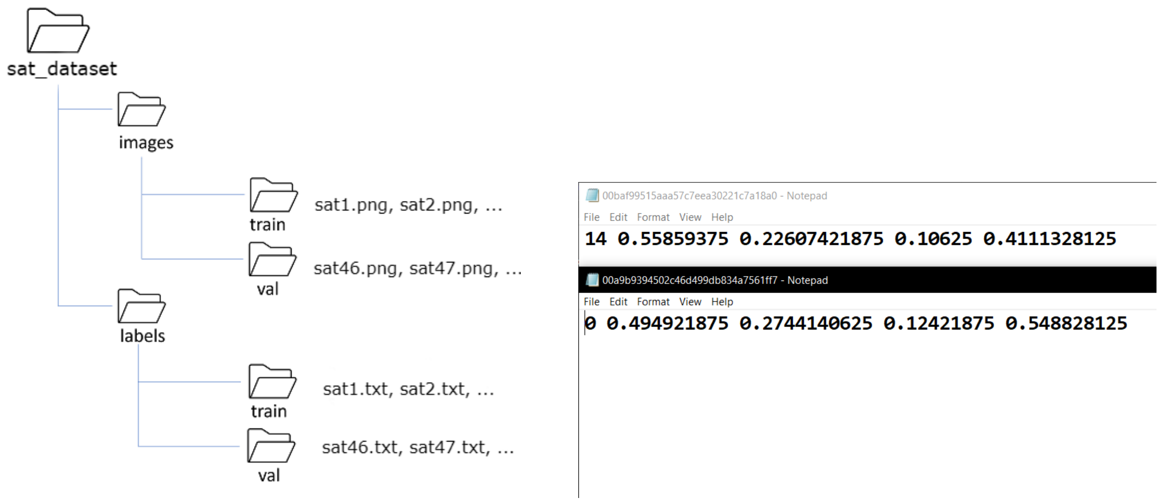


Figure 4.2: Original image on the left, augmented trough blur and random flip on the right.

4.2.2 Image Classification – Object Detection

The dataset for image classification was organized into a structured directory format, dividing the data into training, validation, and testing subsets. Each subset contained subdirectories named after the respective classes, with each subdirectory holding the images corresponding to that class. This hierarchical organization enables seamless integration with PyTorch's *ImageFolder* utility, which automatically maps images to their respective class labels based on the subdirectory names. Moreover, to optimize data handling and ensure efficient model training, PyTorch's *DataLoader* was employed. The DataLoader not only facilitates the batching of images but also enables data shuffling to prevent the model from learning patterns in the dataset's order, and supports parallel data loading for improved computational performance. Here is also where all the above discussed data augmentation takes part, alongside normalization, and conversion to tensor format. All operations applied to our dataset in order to standardize the input data. This systematic approach ensures robust and efficient data pipeline, enabling consistent preparation and feeding of images into the classification system during all phases of the workflow: training, validation, and testing.

The dataset for object detection was prepared following the YOLO format, which requires both images and annotations. Images were organized into training, validation, and testing directories, and each image had a corresponding text file containing annotations. These annotations specified the object class and the bounding box details, including the normalized center coordinates, width, and height of each box relative to the image dimensions. Unlike classification datasets, where labels are inferred from folder names, object detection relies on these precise annotations to identify object locations. Images were resized to a fixed size (e.g., 640x640 pixels) required by YOLO, while keeping the bounding boxes proportional. Data augmentation techniques such as flipping, scaling, and rotation were applied during training to make the model more robust. In addition to images and annotations, a YAML configuration file needs to be created to define the dataset structure. This file specifies the paths to the training and validation directories, the number of classes, and the class names. Figure 4.3 shows the directory structure and annotations files.


sat_dataset
images
train
sat1.png, sat2.png, ...
val
sat46.png, sat47.png, ...
labels
train
sat1.txt, sat2.txt, ...
val
sat46.txt, sat47.txt, ...
00baf99515aaa57c7eea30221c7a18a0 - Notepad
File Edit Format View Help
00a9b9394502c46d499db834a7561ff7 - Notepad
File Edit Format View Help

```
14 0.55859375 0.22607421875 0.10625 0.4111328125
```

```
0 0.494921875 0.2744140625 0.12421875 0.548828125
```

Figure 4.3: Data structure for YOLO.

Chapter 5

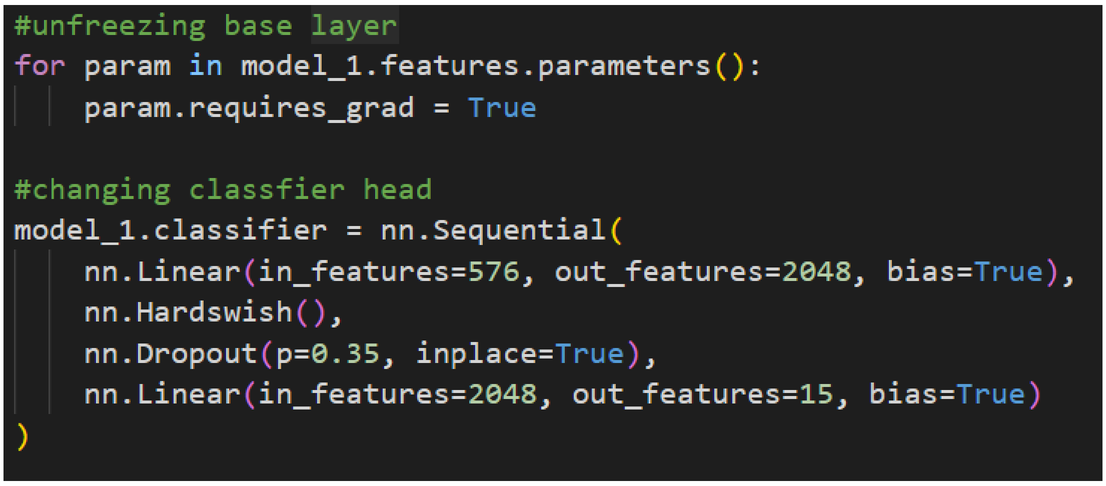
Results

5.1 Training MobileNetV2 for Classification

5.1.1 Training Pipeline

As mentioned in Section 3.3, for the classification task the chosen model is MobileNetV3. To prepare the model for training on our dataset, we first initialize the model, load its pretrained weights, and apply the necessary transforms. A critical step is unfreezing the base layers to allow the weights to be updated during training. Without this adjustment, the model would retain the pre-trained weights, limiting its ability to adapt to the specific features of our dataset and ultimately resulting in poor performance.

The next step involves modifying the classifier head of the model to match the requirements of our task as shown in Figure 5.1. This classifier is composed of two fully connected layers, an activation function, and a dropout layer, each designed to enhance learning and prevent overfitting. The first fully connected layer, defined as *nn.Linear(in_features=576, out_features=2048, bias=True)*, transforms the input from 576 features into a 2048-dimensional space, increasing the model's capacity to learn complex patterns. The inclusion of bias=True ensures that a bias term is used to adjust the output of the neurons, further enhancing the model's flexibility. Next, the *nn.Hardswish()* activation function introduces non-linearity into the model, enabling the model to capture complex relationships within the data by learning non-linear transformations of the input features. To address overfitting, a *Dropout* layer is included. Dropout is a regularization technique that randomly sets 35% of the input units to zero during training, which helps the model generalize better by reducing the likelihood of overfitting. The final component of the classifier is another fully connected layer, defined as *nn.Linear(in_features=2048, out_features=15, bias=True)*. This layer maps the 2048 features to 15 output classes, corresponding to the number of classes in our dataset. The output of this layer represents the *logits*, or raw class scores, which are used to make predictions. To ensure the model operates efficiently on the available hardware, the command *model_1 = model_1.to(device)* is used to transfer the model to the specified device. In this case, the training is performed on a Google Compute Engine backend using a Tesla T4 GPU with 15 GB of RAM.



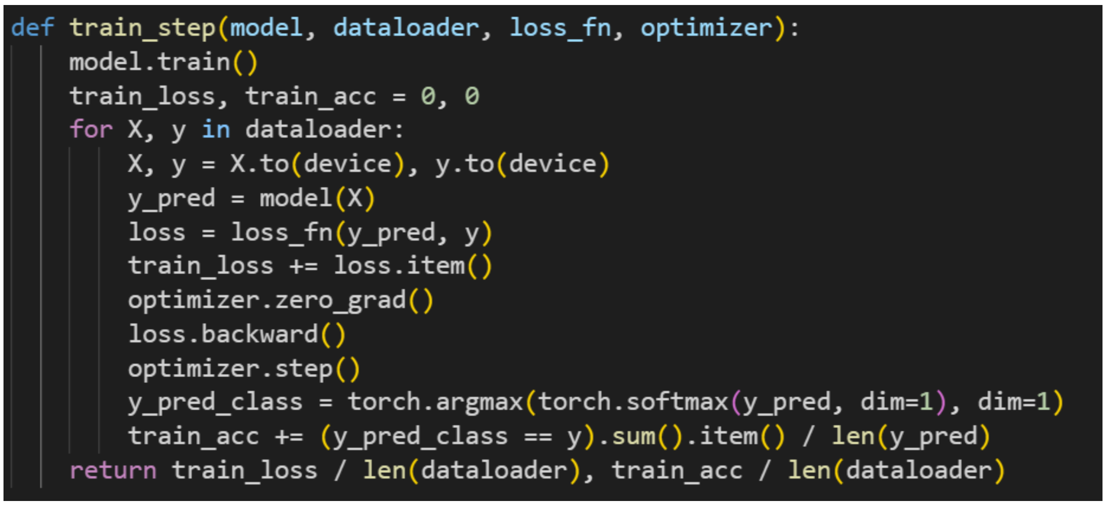
```python
#unfreezing base layer
for param in model_1.features.parameters():
    param.requires_grad = True

#changing classfier head
model_1.classifier = nn.Sequential(
    nn.Linear(in_features=576, out_features=2048, bias=True),
    nn.Hardswish(),
    nn.Dropout(p=0.35, inplace=True),
    nn.Linear(in_features=2048, out_features=15, bias=True)
)
```

Figure 5.1: Code snippet of the model setup.

Through extensive experimentation parameters were chosen as follows. Stochastic Gradient Descent (SGD) emerged as the optimal choice as the optimizer. Providing better generalization. A learning rate of 0.1 was identified as the most effective for this project, enabling the model to achieve robust performance. Smaller learning rates led to slower training and occasionally suboptimal results, while larger values caused oscillations around the minimum. The chosen value provided a balanced trade-off between convergence speed and stability. For this classification task, where the goal is to correctly assign one of 15 classes to each input, the Cross-Entropy Loss function was used being particularly suited to multi-class problems, as it penalizes incorrect predictions more heavily and rewards confident, correct classifications. By doing so, it encourages the model to output probabilities that are as close as possible to the true class distribution.

This carefully selected combination of SGD, a lr of 0.1, and Cross-Entropy Loss ensured that the training process was both efficient and effective, resulting in a model that generalized well to unseen data after extensive fine-tuning.

The last thing to implement are the training and testing functions. These are essential components for training and evaluating a neural network model using PyTorch. The *train_ step* (Figure 5.2) function defines the operations performed during a single epoch of training, including setting the model to training mode, performing forward passes to compute predictions, calculating the loss, updating the model's parameters through backpropagation, and tracking the training loss and accuracy. The *test_ step* function, on the other hand, evaluates the model's performance on a validation or test dataset. It sets the model to evaluation mode to prevent updates to parameters and computes metrics such, loss and accuracy without calculating gradients. This is an important aspect, since accuracy on the train set is usually low, especially in the first epochs, the model would update its parameters based on wrong clues.



```python
def train_step(model, dataloader, loss_fn, optimizer):
    model.train()
    train_loss, train_acc = 0, 0
    for X, y in dataloader:
        X, y = X.to(device), y.to(device)
        y_pred = model(X)
        loss = loss_fn(y_pred, y)
        train_loss += loss.item()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        y_pred_class = torch.argmax(torch.softmax(y_pred, dim=1), dim=1)
        train_acc += (y_pred_class == y).sum().item() / len(y_pred)
    return train_loss / len(dataloader), train_acc / len(dataloader)
```

Figure 5.2: Code snippet of the train step.

5.1.2 Results

Training results with the model set up accordingly are shown in Figure 5.3. This case serves as a perfect demonstration of overfitting - at first glance, these results might be interpreted as successful as train accuracy reaches over 80% only 30 epochs. However, in reality, the model has learned the specific training dataset too thoroughly, resulting in very poor generalization power on unseen images. This is evidenced by the testing accuracy, which fluctuates around 25%. To the author's best knowledge, many anti-overfitting techniques have been implemented with poor results. This is further validated by the quite high dropout rate of 35% that we set during tuning. Moreover, the test loss continuously increases, even reaching values of 8, which is exceptionally high. In the following sections, we will demonstrate how these graphs should look when the model is performing well, particularly after implementing object detection.

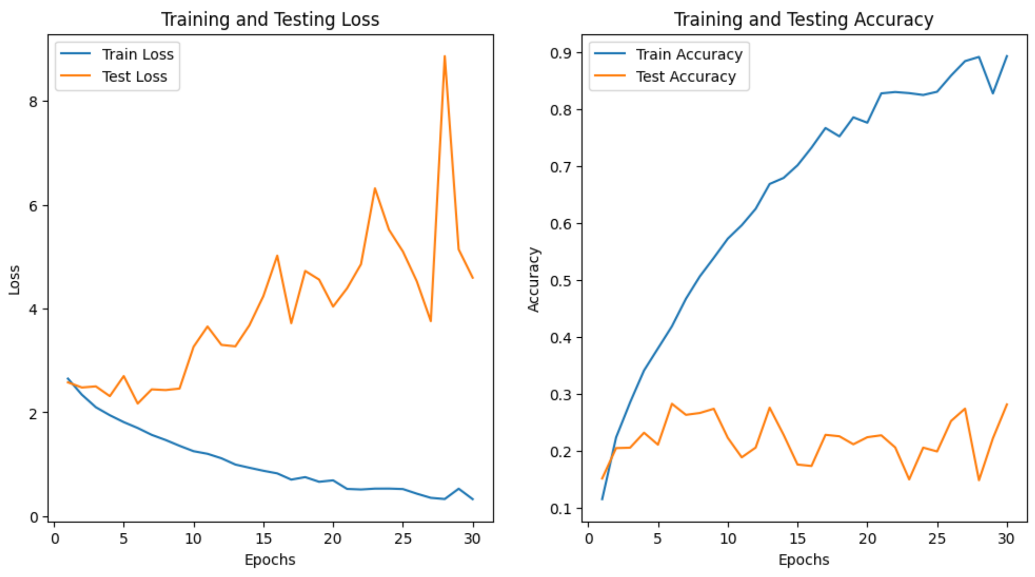


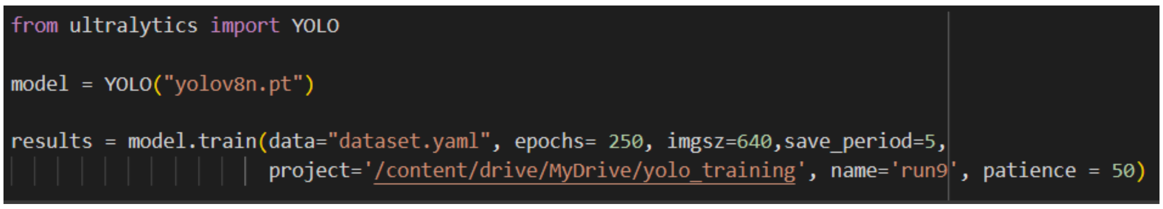
Figure 5.3: Training results over 30 epochs using only image classification.

5.2 Training YOLOv8 for Object Detection

5.2.1 Training Pipeline

The training pipeline for YOLO is designed to operate almost like an API, which makes it a popular choice for object detection tasks. The process begins with specifying the model configuration, where a YOLO variant needs to be selected based on the trade-off between speed and accuracy. In our case YOLOv8n will be the model, where the "n" stands for "nano" indicating the smallest and most lightweight version of the YOLOv8 model. More accurate models are also available, at the cost of inference speed and computational power usage. The model uses pre-trained weights to leverage transfer learning, allowing the model to benefit from prior knowledge gained from large datasets.

The training is initiated through a single command, specifying parameters such as the model architecture, the dataset .yaml configuration file, the number of training epochs, and image resolution (Figure 5.4). During this phase, YOLO automatically handles tasks such data loading, applying data augmentations, computing the loss, and updating model weights using backpropagation. Training progress is monitored through built-in logging and visualization tools that provide insights into metrics such as training and validation loss, mean Average Precision (mAP), and real-time evaluation of predictions.



```python
from ultralytics import YOLO

model = YOLO("yolov8n.pt")

results = model.train(data="dataset.yaml", epochs= 250, imgsz=640,save_period=5,
                      project='/content/drive/MyDrive/yolo_training', name='run9', patience = 50)
```

Figure 5.4: YOLO training setup.

5.2.2 Accuracy Metrics

Precision metrics are essential tools for evaluating the performance of object detection models. They quantify the accuracy and reliability of predictions by assessing how well a model identifies objects and their corresponding locations. These metrics provide a comprehensive view of detection quality, balancing factors such as overlap, correctness, and confidence. Below are the key precision metrics commonly used in object detection:

- **Precision and Recall:** Precision measures the proportion of correctly identified objects among all predictions, while recall evaluates the ability to find all relevant objects. Together, they provide a balance between prediction quality and completeness. Predictions are categorized into four outcomes based on their accuracy: True Positives (TP), False Positives (FP), False Negatives (FN), and True Negatives (TN). **True Positives** occur when the model correctly identifies an object, with its predicted bounding box sufficiently overlapping the ground truth. In contrast, **False Positives** represent incorrect detections, where the model predicts an object that does not exist or fails to meet the required overlap with a true object. **False Negatives** arise when the model misses detecting an actual object, leaving a ground truth object unaccounted for. Lastly, **True**

Negatives are predictions where the model correctly identifies the absence of an object in areas with no ground truth annotations.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \qquad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{5.1}$$

- **Intersection over Union (IoU):** Measures the overlap between the predicted bounding box and the ground truth bounding box, calculated as the area of their intersection divided by the area of their union. It assesses alignment accuracy. Graphic example is shown in Figure 5.5.

$$IoU = \frac{\text{area of overlap}}{\text{area of union}} \tag{5.2}$$

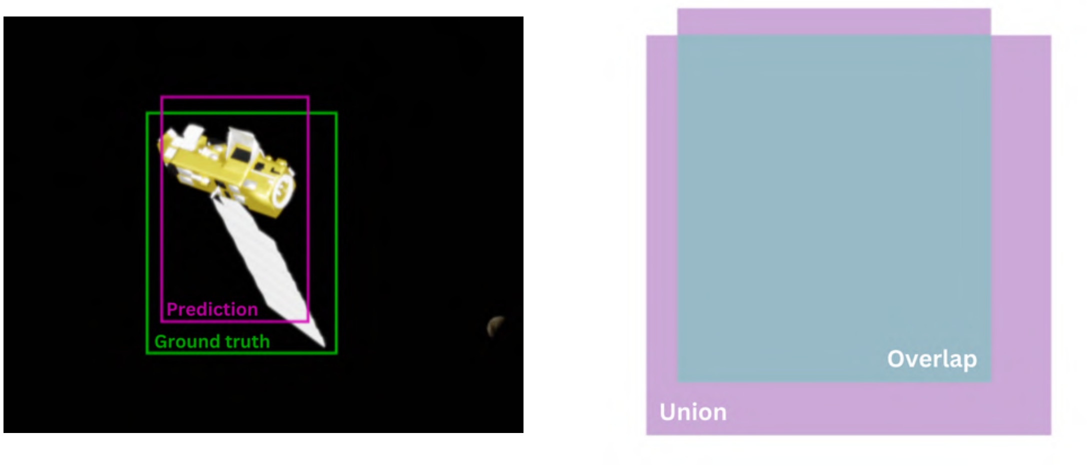


Figure 5.5: Data structure for YOLO.

- **Average Precision (AP):** A fundamental metric that combines precision and recall across different confidence thresholds. It condenses the Precision-Recall curve into a single score for each class.

- **Mean Average Precision (mAP):** Extends AP to multi-class detection by averaging AP scores across all classes. It considers multiple IoU thresholds to provide a holistic assessment of overall model performance. **mAP@50** calculates the mean average precision with an IoU threshold of 0.50 (50%), meaning a predicted bounding box is considered correct if it overlaps with the ground truth by at least 50%. **mAP@90** a stricter IoU threshold of 0.90 (90%), requiring a much closer match between the predicted and ground truth bounding boxes.

5.2.3 Results

Training of YOLOv8n was performed for 250 epochs, exhausting any chance of significant % points improvement. Results shown in Table 5.1 refer to the testing unseen dataset containing a total of 1151 images. The first row represents the average results across all images, some of them, for example the mAP@90 seem to be quite low with only 72% accuracy, but it's imperative to remember that the mean average precision @ 90% is the most stringest accuracy metric, and in reality, especially for the nano model, it is a remarkably impressive achievement. For instance, the mAP@50 has a

Class	Images	Instances	P	R	mAP@50	mAP@50-95
All	1151	1151	0.834	0.795	0.860	0.724
2	56	56	0.931	0.958	0.986	0.831
3	55	55	0.880	0.836	0.910	0.735
5	68	68	0.814	0.853	0.899	0.865
6	70	70	0.779	0.843	0.866	0.806
11	72	72	0.918	0.917	0.968	0.790
13	59	59	0.826	0.483	0.732	0.453
14	67	67	0.784	0.868	0.923	0.864
18	72	72	0.762	0.778	0.818	0.729
19	81	81	0.729	0.764	0.778	0.618
20	85	85	0.809	0.750	0.803	0.686
22	78	78	0.869	0.853	0.927	0.760
24	105	105	0.843	0.600	0.722	0.453
25	91	91	0.923	0.923	0.952	0.880
28	100	100	0.874	0.950	0.977	0.929
30	92	92	0.776	0.554	0.636	0.459

Table 5.1: Performance metrics across different classes (named after numbers).

86% accuracy. For many real-world applications, 80%+ is considered sufficient for practical use.

These results can be interpreted further using simplistic yet rational reasoning. This can be done by taking the best and worst results of a specific metric–in this case mAP@50–and compare the images that each class contains, respectively class 2 at 98.6% and class 30 at 63.6%.

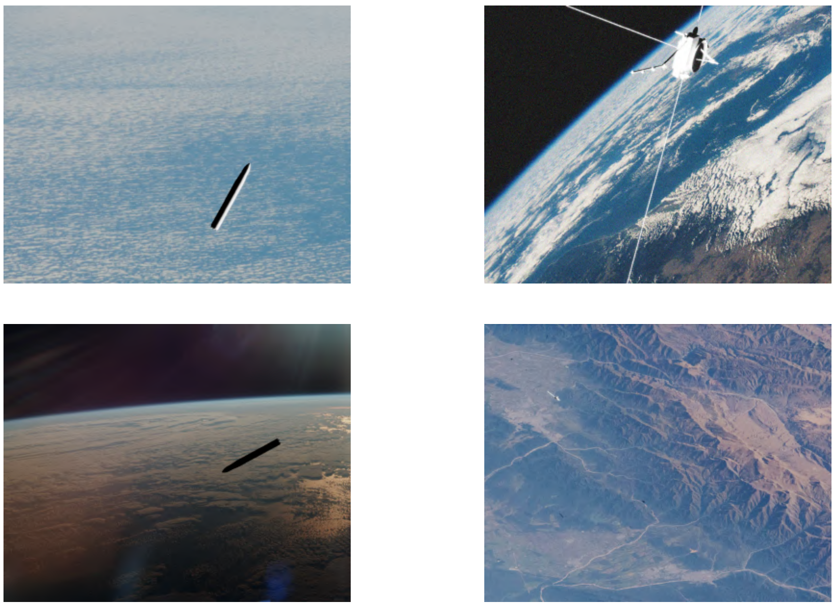


Figure 5.6: Spacecraft from class 2 on the left, class 30 on the right.

It is evident from Figure 5.6 that, the spacecraft on the left has a much more distinguishable shape than the one on the right. In the fourth image, the spacecraft is barely noticeable, making it particularly challenging to create an accurate bounding box that accounts for its very long, thin antennas extending from a prominent and easily detectable body. Given these characteristics, it is clear why the model struggled to predict correct bounding boxes around this particular spacecraft.

5.3 Implementing Object Detection to Enhance Classification Accuracy

This subchapter demonstrates the effectiveness of implementing a two-stage object detection and classification model, demonstrating its significant advantages over standalone classification approaches. In real-world scenarios, attempting to classify spacecraft directly from full images would be suboptimal, as the target often appears as a small object against vast, complex backgrounds. The more efficient approach is first, to identify the region of interest (RoI) by deploying ODN, then crop and resize the image and finally perform the classification. When training a classification model, it's important to understand that the model doesn't comprehend what a spacecraft truly is; rather, it learns to recognize abstract patterns common across training images. This becomes problematic when dealing with images where the spacecraft occupies only a few pixels against a prominent–for example–Earth background. The model struggles to identify the relevant features among the overwhelming background information. Our approach addresses this limitation by employing object detection to locate the spacecraft within the image. By cropping around the detected bounding box, we effectively eliminate a substantial portion of the background noise.
The process begins by loading the YOLO model, input images, and ground truth data, followed by creating an output directory for storing cropped images. For each image, the model detects spacecraft and generates bounding boxes, which are then compared to the ground truth using the IoU metric. A detection is considered correct if its IoU exceeds a defined threshold (0.5). For matched detections, the corresponding regions of the image are cropped and saved in a class-specific directory, facilitating data handling for image classification for further analysis. Images where no spacecraft is detected are simply skipped and will not be part of the final cropped dataset. Table 5.2 demonstrates an accuracy of 93.18% in detecting spacecrafts with our pre-trained YOLOv8n model.

Figure 5.7 shows the output of the detect & crop process. These images are not ready yet to be fed to the classification model as they come with different sizes, making them unsuitable for training since our model expects fixed input shapes. This problem gets addressed by the transforms applied in the dataloader, normalizing all images to a 224x224 shape. With all images separated accordingly for each phase of the pipeline, the process illustrated in Section 5.1.1 was repeated.

	Fed to YOLO	IoU $>$ 50%	IoU $<$ 50%	Not detected
Number of Images	8021	7474	247	304

Table 5.2: Out of the 8021 images fed to the model 7474 were correctly detected, 247 had an intersection lower than 50% and no spacecraft was detected in 304 images.

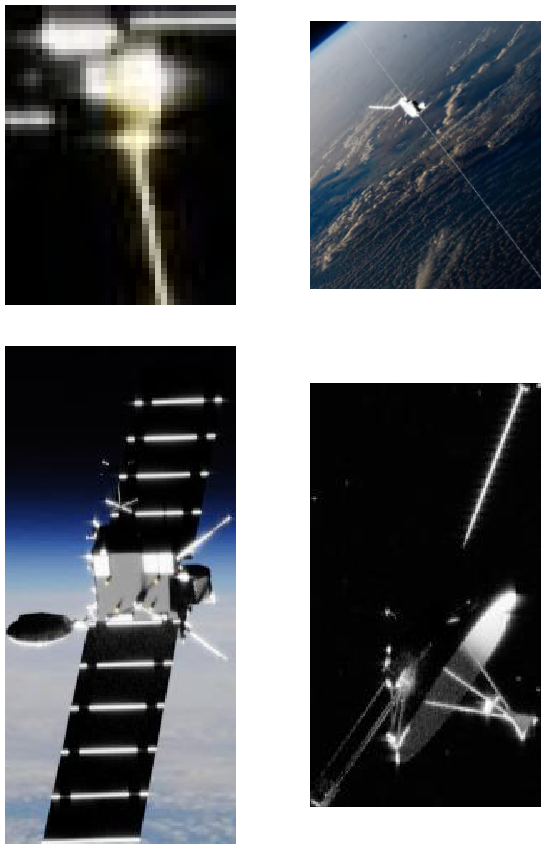


Figure 5.7: Samples of the cropped dataset.

Figure 5.8 shows the new performance. After only 30 epochs the model converges to an optimal training accuracy of 99%+ and acceptable testing accuracy of around 85%. It is worth noting that these results were reached as soon as the seventh epoch, with more steady improvements in later epochs.

Unlike the other case when MobileNet was used by itself, here, the model still does show room for even more improvement. Being this the case, the training was later performed across a total of 260 epochs. Given Google Colab usage limit this was performed in different sessions by saving checkpoints of the model status and then resuming training when resources were available again. The learning rate was steadily manually lowered, in total by around a order of magnitude every 100 epochs, on top of that a weight decay of 1e-4 and a momentum of 0.9 were also implemented. As the model converges to the minimum the step towards that direction should get finer and finer to not overshoot, thus ending up with a lr = 0.005.

After 230 more epochs train accuracy stabilized on 100%, consequently train loss got as low as 0.002. Test accuracy got refined by around 7% dropping the average test loss from 0.61 to 0.397. Figure 5.9 shows only the last 80 epochs (from 180 to 260) as it was the last checkpoint of training. Across these 80 epochs, with the model converging to the limit, there was an improvement of almost 2% of accuracy. Even though the train curves still show some inclination, there is almost nothing else to squeeze out of it performance wise, if training continues that trend will get flatter and flatter asintotically.

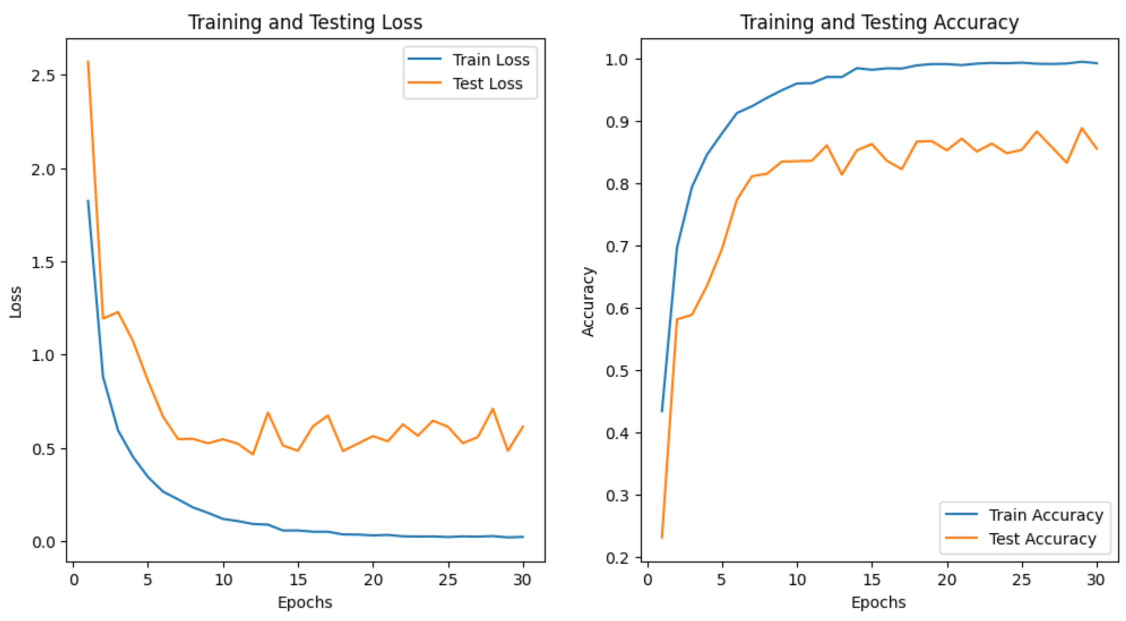


Figure 5.8: Training results over 30 epochs with object detection implemented.

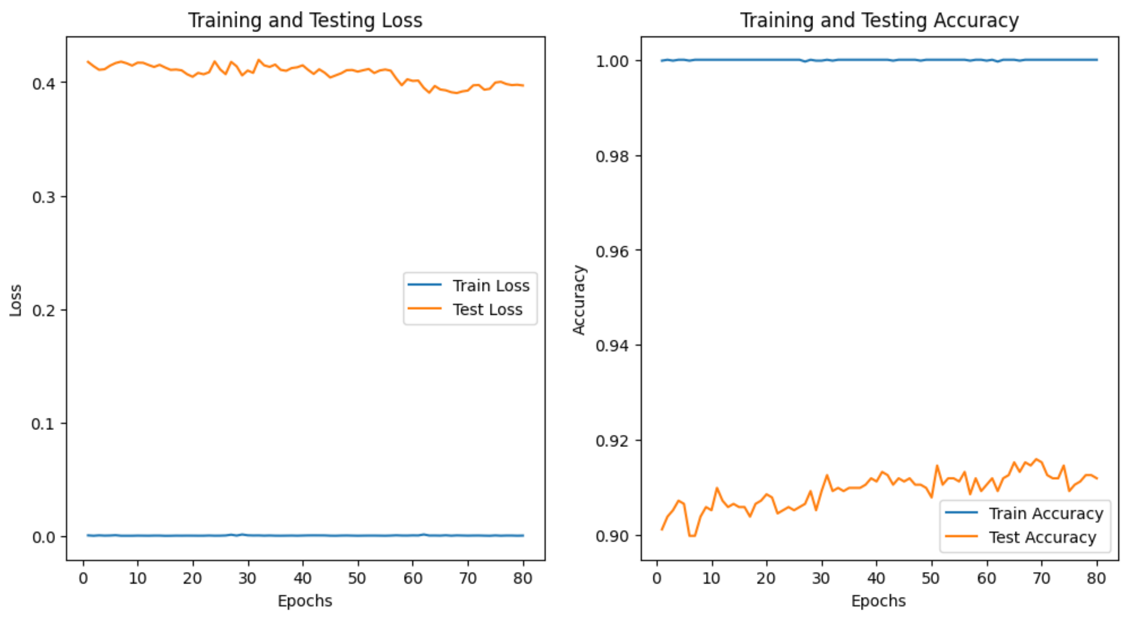


Figure 5.9: Train and test curves over the last 80 epochs out of 260.

5.4 Feature Maps

Chapter 2.3 served as a broad introduction on how a CNN interprets the input image and how it processes it. This section will expand on those concepts giving a better understanding on what actually happens to the image once its fed to the model. Firstly it shall be recalled that feature maps play a fundamental role as intermediate representations that capture essential visual information from the input image. As an image passes through the layers of a convolutional neural network (CNN), each layer generates feature maps that represent increasingly abstract characteristics of the original image. The initial layers typically detect basic elements such as edges, textures, and color patterns, while deeper layers combine these elementary features to recognize more complex structures and object-specific attributes. These feature maps can be conceptualized as learned filters that highlight particular aspects of the input image that are relevant for the classification task. The dimensionality and nature of these features evolve throughout the network: from low-level spatial features in early layers to high-level semantic features in deeper layers, effectively transforming the raw pixel data into a rich, hierarchical representation that enables accurate classification. Understanding how these feature maps develop and interact is crucial for comprehending the internal mechanics of CNN-based classification systems. Going back to Figure 2.2 we can now clearly distinguish each block, the last 2 orange sort of lines represent the 2 fully connected layers of the classifier head, the same ones set up in Figure 5.1. All blocks prior the classifying fully connected layers are all responsible for image processing and feature extraction. The number of layers in modern architectures is usually much greater than those used for the classification head. MobileNetV3 has a total of 13 layers and 2542856 parameters that together compone the backbone of the model. The full strucutre is visualizable by the command *model.features*, it shows the comprehensive architecture, with the size of kernels, convolutional matrices, batch normalization, non linear functions etc.

With the architecture clear in mind, we can now access each layer and extract what the model "sees", getting an insight of how the input image is being processed. Again, it is expected that accessing the first layers will result in images similar to the input one, with a slightly lower size-resolution. In contrast, accessing deeper layers should output images that are only a few pixels of resolution and resemble abstract features.

The feature maps represented in Figure5.10 from the top, refer to the second, fourth and tenth layer respectively. Rows represent 5 different input images, columns are the 16 different channels (filters) that each convolutional layer has. As already mentioned, each filter is responsible for detecting different patterns troughout the same image. Each activation map, if the color is yellow/green indicates strong activations, where the filter detected patterns it was trained to recognize, in contrast, dark (blue) areas indicate low or no activation, meaning the filter did not find relevant patterns in those regions.

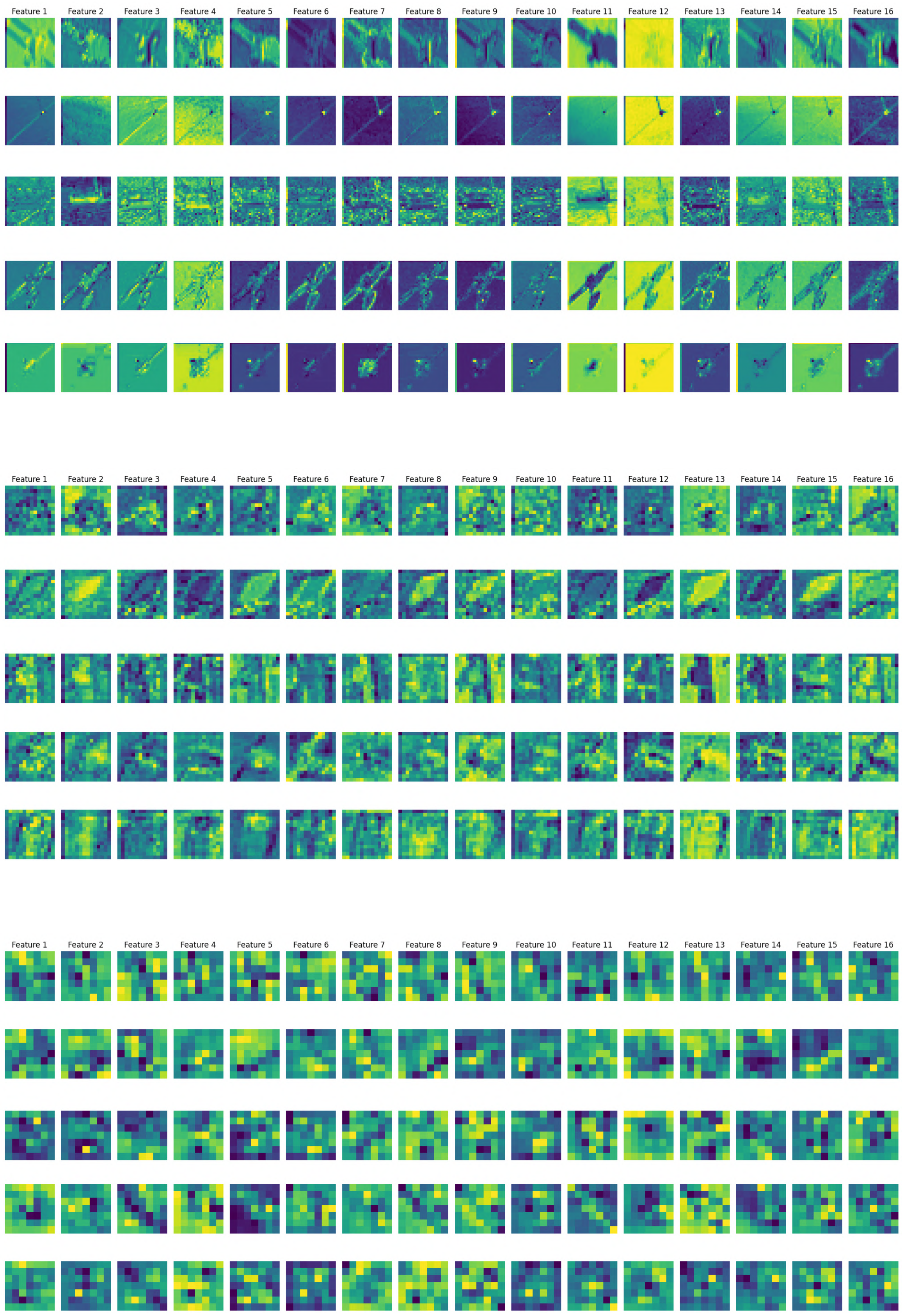


Figure 5.10: Samples of the feature maps from the cropped dataset.

5.5 Embedded Spaces

Embedding spaces provide a powerful lens through which we can examine how neural networks interpret and distinguish between different classes of images. These high-dimensional spaces transform complex visual inputs into compact, meaningful representations that capture the essential characteristics of each image. Visualization techniques like t-Distributed Stochastic Neighbor Embedding (t-SNE) [12] offer a window into these complex representations, allowing to map high-dimensional data into two- or three-dimensional spaces. However, it is fundamental to approach these visualizations with a clear understanding.

It has been explained that the input image is processed as a matrix of different dimensions. At the end of the feature extractor backbone, the matrix reaches a convoltuted complexity, as all the the features extracted by each different channel are put all together into a single entity. Ready to be fed to the classifier head, the matrix gets changed in shape, more in particular it needs to become a one single column with as many rows as the first input layer of the classifier head has, noting, once again,that the number of neurons is the one seen in Figure 5.3 where each neuron represents a number from 0 (off-no activation) to 1 (on-fully active) mapped from the numbers that composed the matrix. What gets created is de-facto a column vector, and vectors have the intrinsic property of belonging to a vector space, thus making it representable. This is the same concept of the 2/3D cartesian plane with the difference of being in 576 dimensions. This is the equivalent of saying that every input image, after some mathematical operations, gets assigned a specific vector in some vector space.

A vector space of 576 dimensions is difficult to imagine and represent for obvious reasons, t-SNE comes in hand by mapping high dimensional spaces to lower dimensional spaces, making it possible to visualize those vectors (images) and the place they occupy relative to each other. Perhpas it is interesting to see if there is any relationship between the space they occupy and the caracteristichs of the images. This mapping can be done, again, at shallow and deeper layers. It could be expected that in shallow layers–where the model has not yet learnt–there wouldn't be much relationship between the image features in relation to the space it occupies and its neighbors; in deeper layers, where the model has progressed learning, perhaps images could start to cluster in some specific spaces.

Figure 5.11 represents the embedding space of a shallow layer, for clarity images have been added to represent better their classes and characteristcs. As expected there seem to be no correlation between how images occupy the embedding space. In a simplistic description, this can be interpeted as, going up or down, left or right in the plane won't make any difference and no stable repeating patterns will be found.

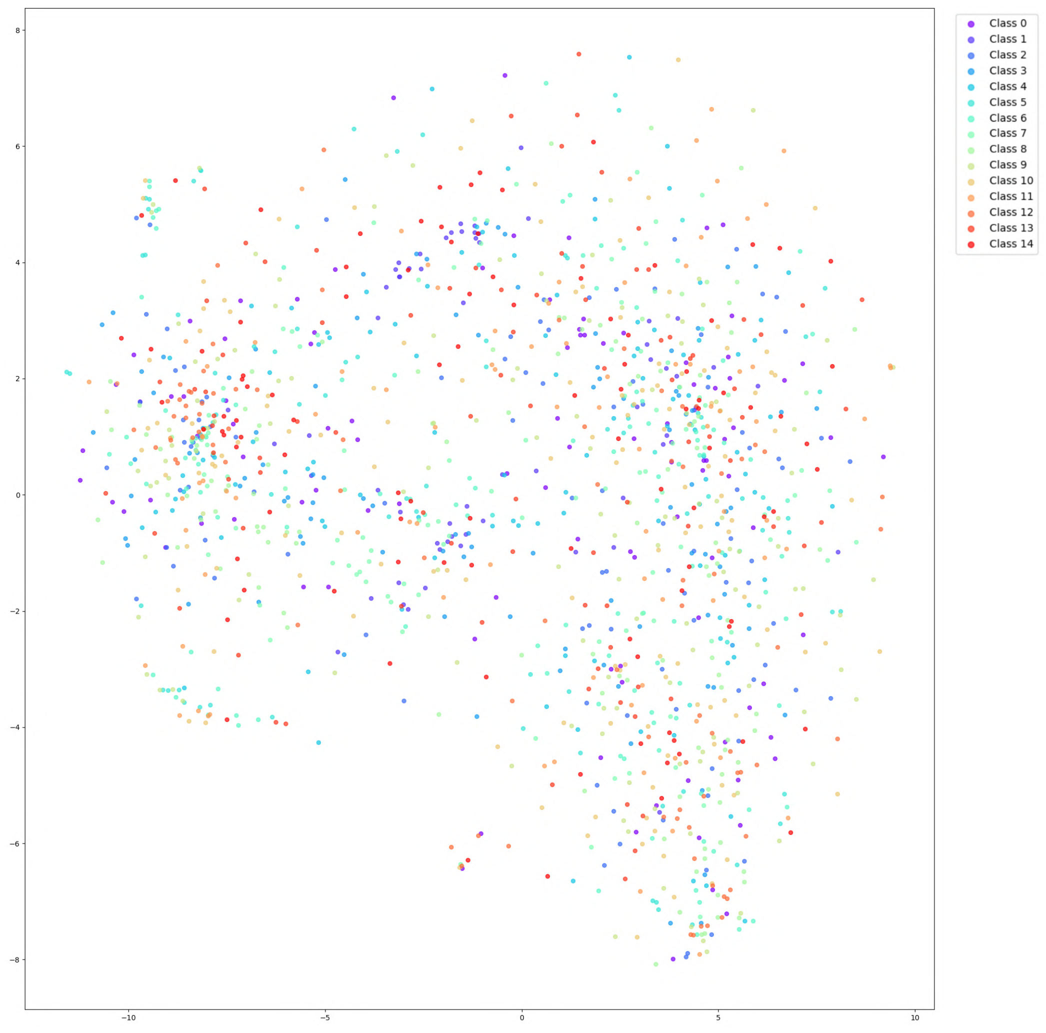


Figure 5.11: Class distribution in a shallow layer.

Getting into a medium layer gives a glimpse of the learning process of the model, as here, still no pattern can be detected class-wise, all images seem to occupy a random space. It is only with the help of representing some images of these vectors that we can see that the model has learnt a pattern, colors. One of the easiest way to distinguish images and find patterns, without further instructions, is perhaps by colors, this is exactly what the model is doing, not realizing this is not the classification it needs to do. Something that will be picked up as soon as more abstract and complex features will be comprehended. The model primarily leverages the features it has learned—such as edges, corners, basic geometric shapes, and color distributions—to classify images. Among these, color appears to be the most prominent distinguishing factor. Additionally, the transition between color-based clusters in the t-SNE embedding space exhibits a smooth gradient. For instance, moving from the center of the embedding space towards the northwest direction corresponds to a progression towards darker images. This indicates that the t-SNE projection effectively organizes the data by capturing variations in fundamental visual features, with a notable emphasis on color transitions. This is shown in Figure 5.12.
Finally, Figure 5.13 shows a deeper layer, where classification is accurate and all classes occupy clearly distinct spaces, making clusters of same classes form. Here, it is more difficult to understand what exactly changes going one direction or another, since patterns that the model picked up at this point are highly abstract and most of the times do not have an interpretable meaning for humans.

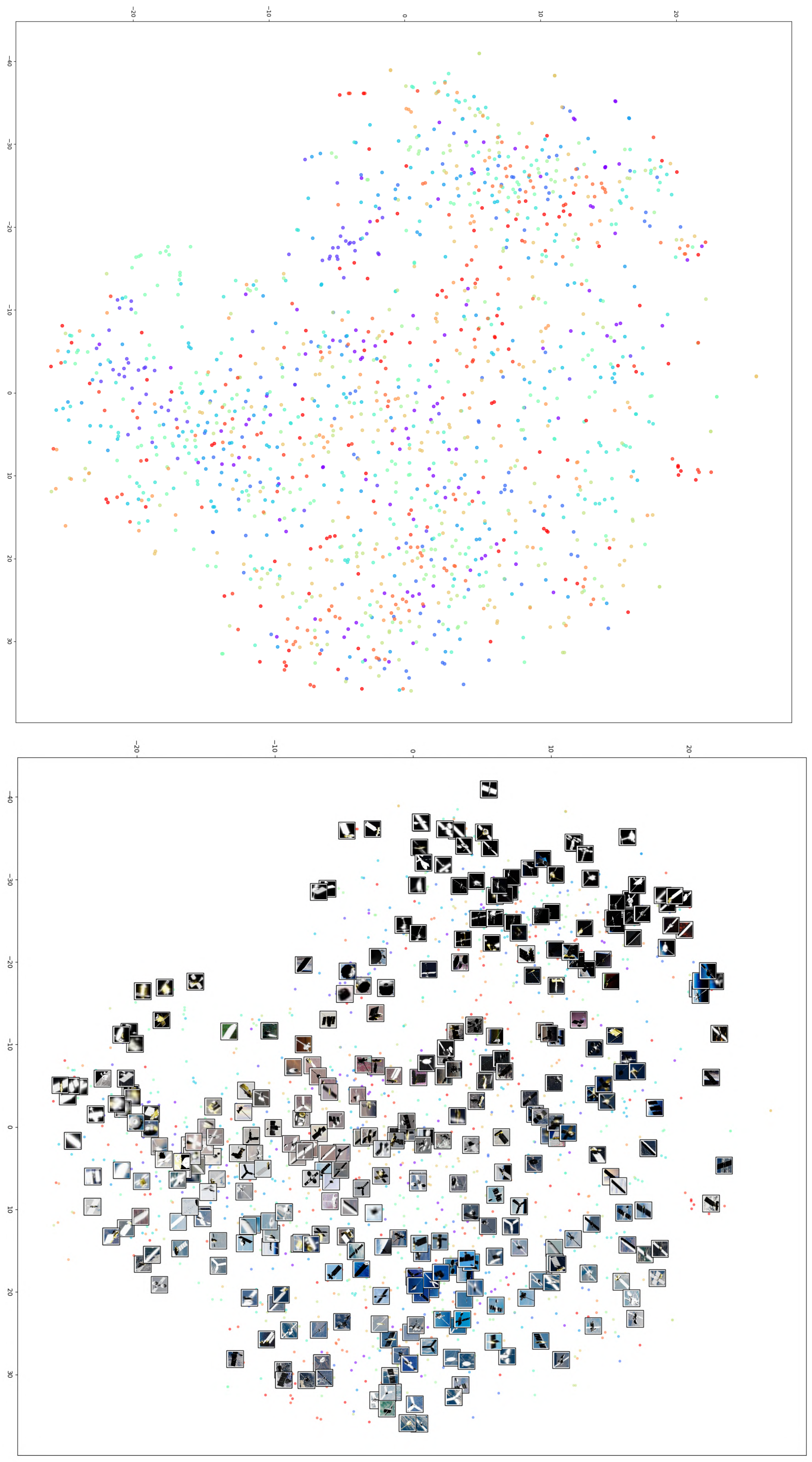
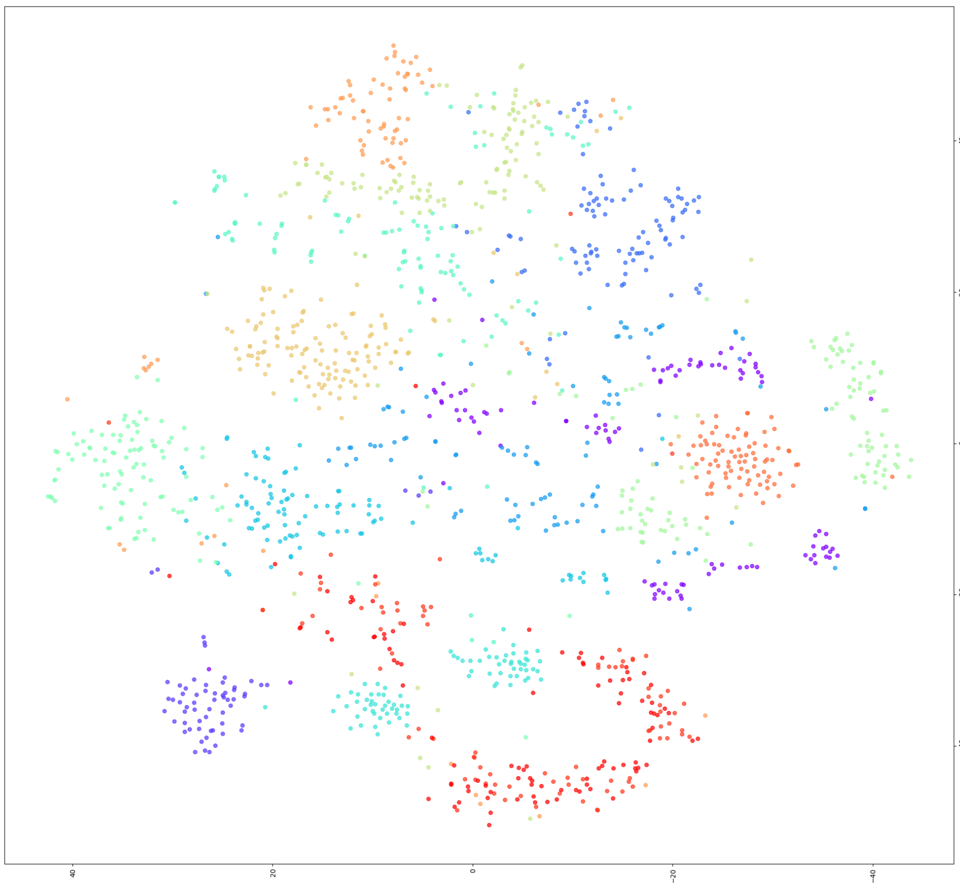


Figure 5.12: Class distribution in a medium-depth layer.



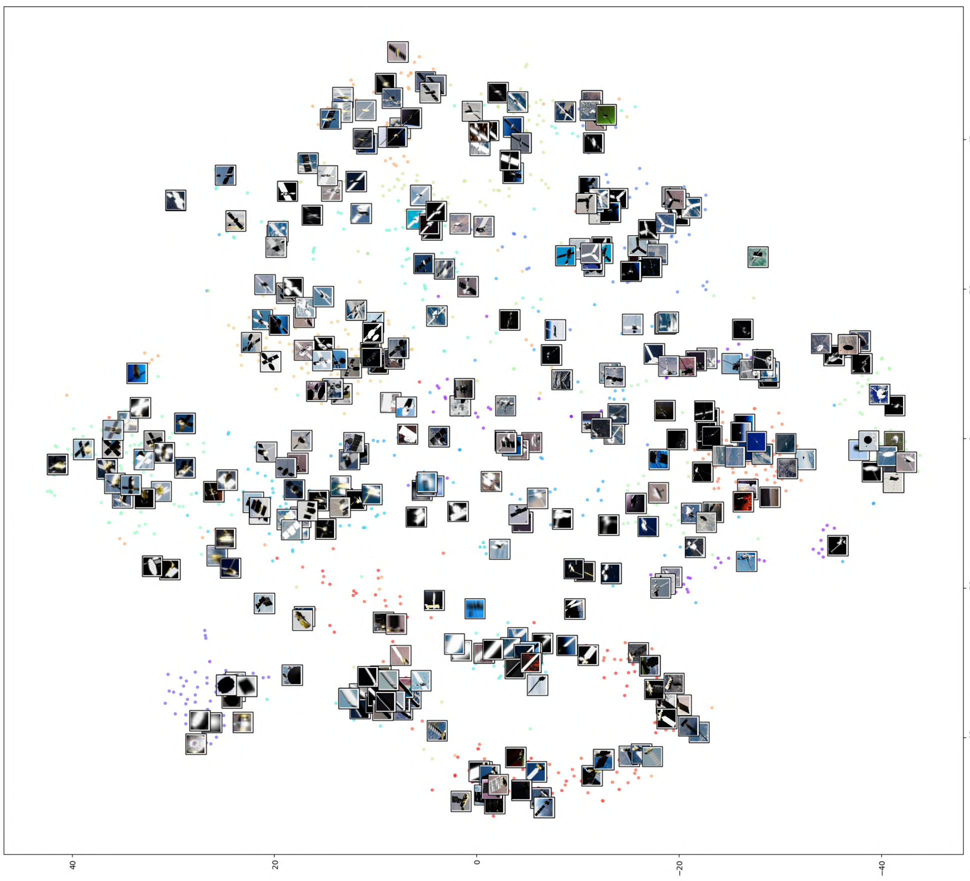


Figure 5.13: Class distribution in a deep layer.

What appears as a clear separation or clustering in a 2D projection can be fundamentally misleading. The dimensionality reduction process inevitably compresses a vast amount of information, potentially obscuring the true relationships between data points. Consider the journey of dimensionality: In our initial 2D representation, certain image classes might appear to overlap, suggesting similarity or classification ambiguity. Yet, as we progressively add dimensions—moving from 2D to 3D, and then continuing to add dimensions until reaching the full embedding space (which in our case encompass 576 dimensions)—a remarkable transformation occurs. Points that seemed indistinguishable in lower-dimensional projections begin to reveal their true spatial relationships. This phenomenon highlights a limitation of human perception. ML models, in contrast, operate in spaces of much higher dimensionality with ease. Each added dimension can dramatically alter the perceived distances and distributions of data points. What might look like a tight cluster in 2D could actually be a complex, spread-out configuration in the full embedding space. By adding just one more dimension, making the plot 3D, this phenomenon can be observed.

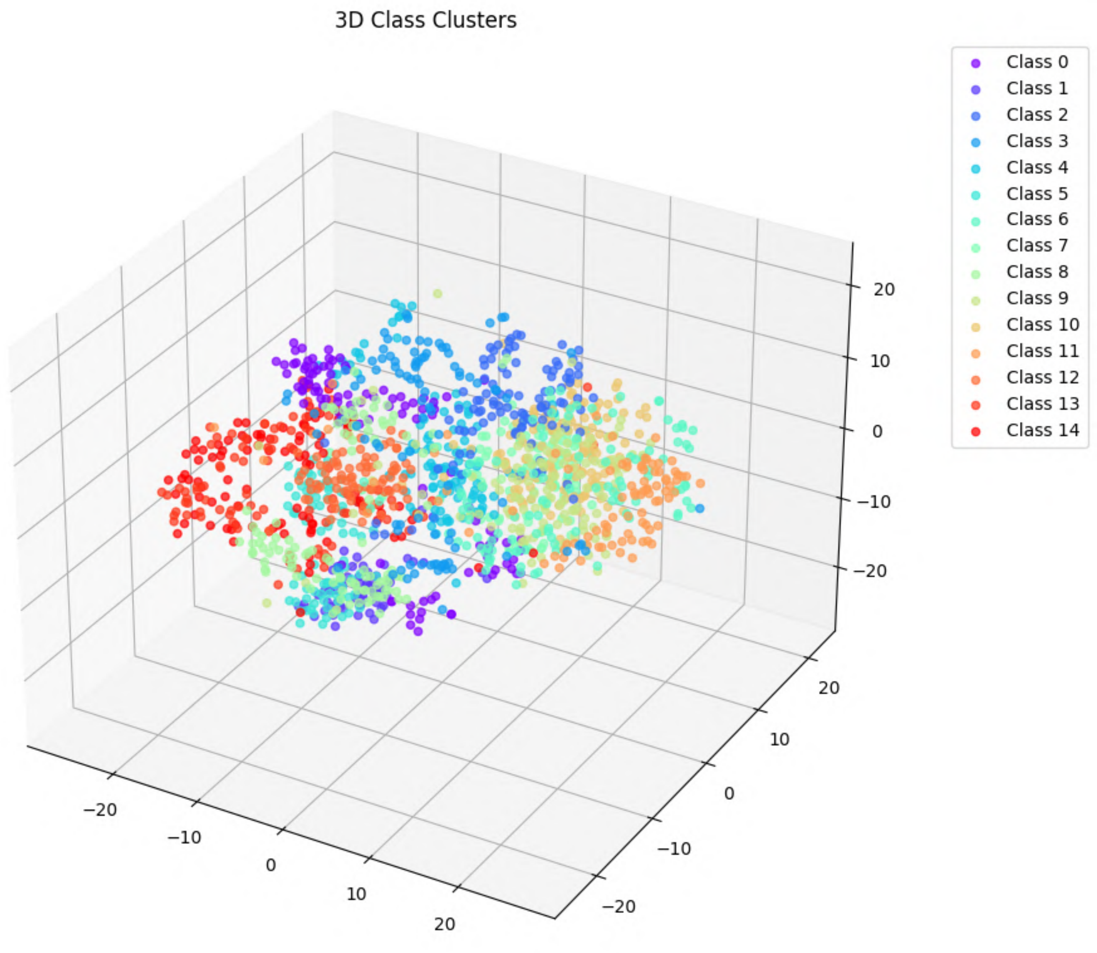


Figure 5.14: 3D t-SNE representation of the embedding space, an interactive version of this plot can be found here.

The top view in Figure 5.15 of the 3D plot resembles the full 2D map shown in Figure 5.13, it can be seen that some regions are indeed over lapping. After careful analysis, for example inside the circled region, by changing persepctive it is assessed how in reality those vectors lie in very distinct regions in the embedding space as shown in Figure 5.16.

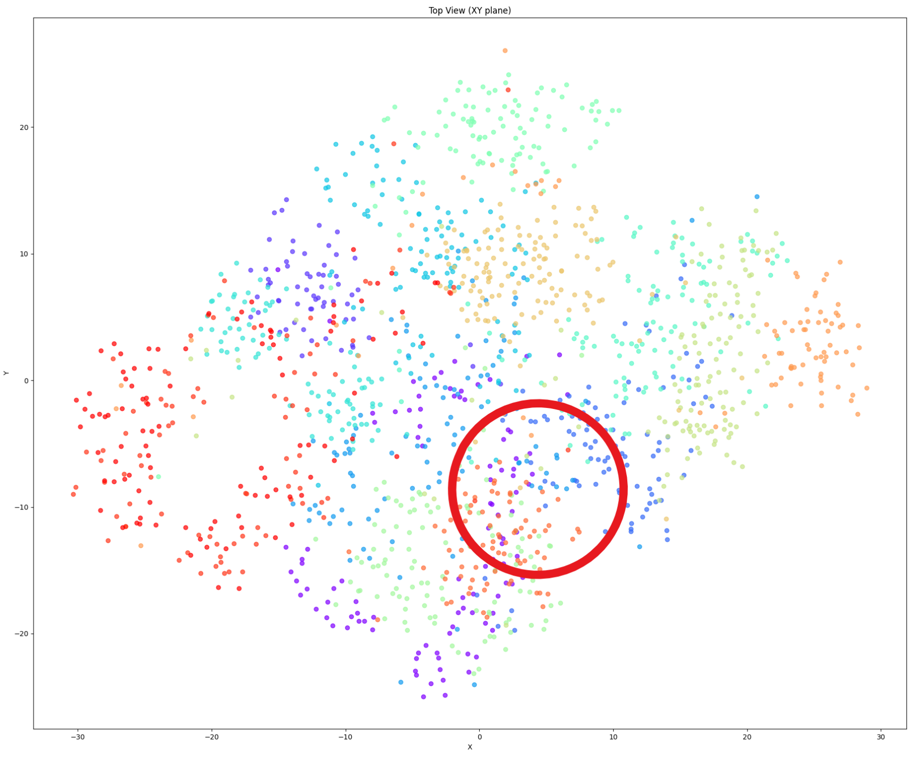


Figure 5.15

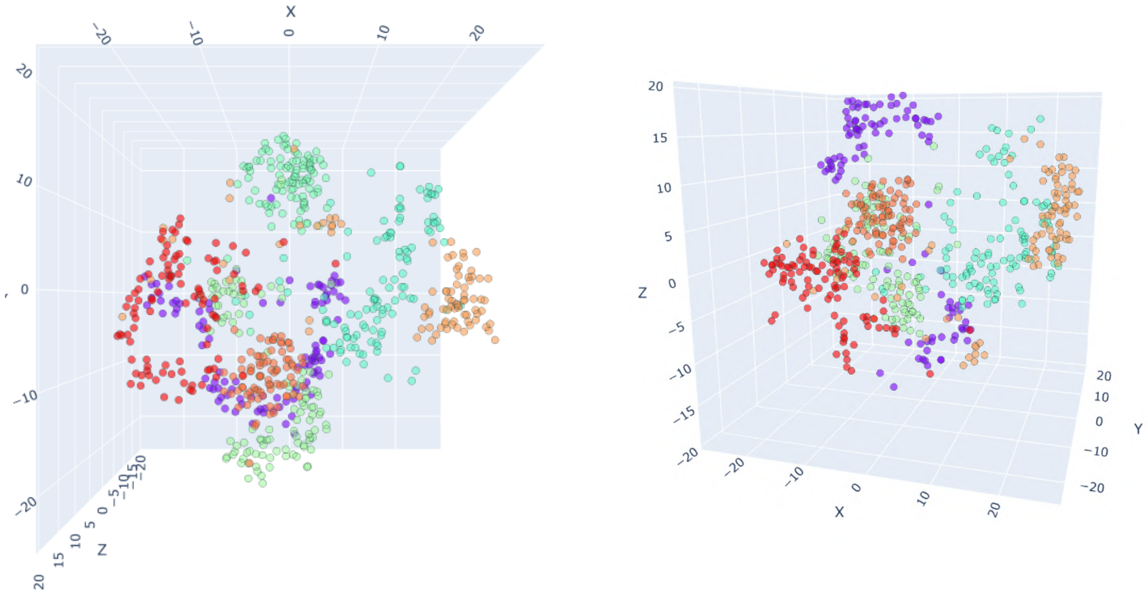


Figure 5.16: Left map is the top view from the 3D plot, the one on the right shows a side view, points that seem clustered in a 2D visualization might be on different levels od depth. This gets acquainted the more dimensions you add.

Chapter 6

Conclusion

The primary objective of this thesis was to develop and evaluate machine learning methodologies for the classification of spacecraft in the PoseBowl dataset, a collection of synthetic imagery designed to simulate realistic conditions encountered in space operations. The study aimed to assess the potential of ML models to address the increasing challenges posed by space debris, focusing on their capabilities, limitations, and overall deployability in real-world scenarios.

Initially, MobileNetV3, a lightweight convolutional neural network, was employed for the classification task. However, the model exhibited significant challenges in handling the complexities of the dataset, which included diverse environmental conditions, varying illumination levels, and partial visibility of spacecraft. The model achieved a testing accuracy that plateaued at approximately 25%, which is indicative of its inability to generalize effectively. Furthermore, the high disparity between the training accuracy, exceeding 80%, and the testing accuracy highlighted severe overfitting issues. These results underscored the limitations of using a lightweight model like MobileNetV3 in isolation for such demanding tasks.

To address these issues, a two-stage approach integrating object detection with classification was proposed. YOLOv8n, the *nano* variant of the YOLO family, was selected for the object detection phase due to its balance of computational efficiency and performance. The model demonstrated exceptional performance, achieving a precision of 0.834 and a mean average precision at 50% Intersection over Union (mAP@50) of 0.86 across all spacecraft classes. This highlights the suitability of YOLOv8n for detecting spacecraft in diverse and challenging scenarios.

The data processing pipeline was then refined based on YOLOv8n's predictions. Cropping was applied around the detected bounding boxes to isolate the spacecraft from their backgrounds, effectively creating a new, high-quality dataset for classification. The processed dataset reduced the complexity of the classification task. When retrained and evaluated using this new dataset, MobileNetV3 exhibited improvement, achieving a classification accuracy of over 92% on previously unseen test images. This significant enhancement validates the efficacy of combining object detection with classification, highlighting the importance of data preprocessing and addressing model limitations.

This study demonstrates that ML, is a viable and effective solution to challenges associated with identifying spacecraft against the rising threat of space debris. The findings emphasize the critical role of robust preprocessing and task-specific model selection in overcoming the limitations of lightweight architectures. Furthermore, the results contribute to advancing autonomous vision-based systems for space applications, paving the way for improved operational safety and debris mitigation strategies.

6.1 Future Work

One significant avenue of improvement is model tuning. While strong performance metrics were achieved, further optimization of hyperparameters—such as learning rates, batch sizes, and augmentation strategies—could enhance accuracy and robustness. Experimenting with advanced architectures, transfer learning techniques, or ensemble models could also provide better generalization, particularly in edge cases like occlusions or challenging lighting conditions.

Another crucial area for development lies in dataset expansion. The current dataset, although comprehensive, is limited in scale and diversity. Future research could involve using tools suchs as Blender to synthetically generate additional images of spacecraft under varied conditions–different lighting, orientations, backgrounds, and textures. By leveraging detailed spacecraft models and simulating realistic orbital environments, researchers could create a significantly larger and more representative dataset. Additionally, domain adaptation methods could help bridge the gap between synthetic and real-world data, improving the real-world applicability of these models.

Finally, an important consideration for future work is the computational constraints of deploying such models on satellite onboard systems. Onboard computers typically have limited processing power and memory, making it challenging to run complex models with high inference times. Future research should focus on optimizing inference time and reducing the computational footprint of models. Lightweight architectures tailored for edge devices, such as satellite computers, should be prioritized to ensure compatibility with the resource-constrained environment of space missions.

Bibliography

[1] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. “Yolov4: Optimal speed and accuracy of object detection”. In: *arXiv preprint arXiv:2004.10934* (2020).

[2] Shan Carter et al. “Activation atlas”. In: *Distill* 4.3 (2019), e15.

[3] ClearSpace. *ClearSpace - A mission to make space sustainable.* `https://clearspace.today/`.

[4] Donald J. Kessler and Bruce G. Cour-Palais. “Collision Frequency of Artificial Satellites: The Creation of a Debris Belt”. In: *Journal of Spacecraft and Rockets* 22.12 (1985), pp. 823–827.

[5] Yann LeCun et al. “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4 (1989), pp. 541–551.

[6] Mohamed Adel Musallam et al. “SPARK: spacecraft recognition leveraging knowledge of space environment”. In: *arXiv preprint arXiv:2104.05978* (2021).

[7] Tae Ha Park et al. “SPEED+: Next-generation dataset for spacecraft pose estimation across domain gap”. In: *2022 IEEE Aerospace Conference (AERO).* IEEE. 2022, pp. 1–15.

[8] Russell P Patera. “General method for calculating satellite collision probability”. In: *Journal of Guidance, Control, and Dynamics* 24.4 (2001), pp. 716–722.

[9] J Redmon. “You only look once: Unified, real-time object detection”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016.

[10] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386.

[11] 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.

[12] Laurens Van der Maaten and Geoffrey Hinton. “Visualizing data using t-SNE.” In: *Journal of machine learning research* 9.11 (2008).