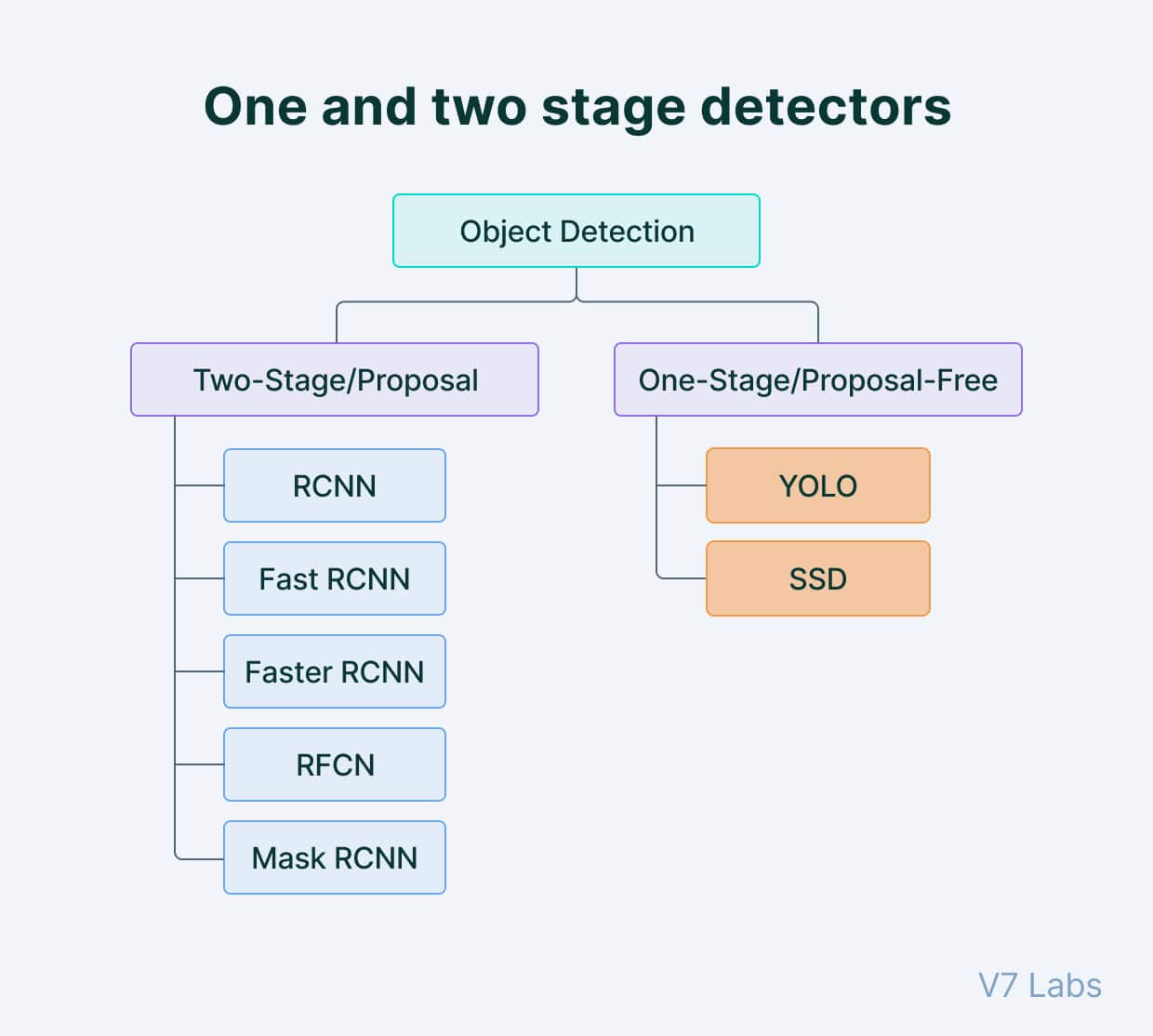
**Perception Tasks:**

**Introduction:**

In these perception tasks, we will first collect a diverse image dataset of artifacts from a simulated environment (Perception 1), which will be used to train a computer vision model. Then, we’ll design and test this model to detect various artifacts (Perception 2). Object detection is a computer vision task to identify and locate objects in images or videos, crucial for applications like surveillance, autonomous driving, and robotics. Detection algorithms fall into two main types: single-shot and two-stage detectors [1].



**Two stage vs single stage models:**

**2 stage model:**

Two-shot object detection makes predictions in two passes: the first generates a set of proposals or potential object locations, and the second refines these proposals and make final predictions, achieving higher accuracy but at a greater computational cost. Single-shot detection is better for real-time needs, while two-shot detection suits accuracy-focused applications.

**Single stage model:**

Single-shot object detection predicts object presence and location in a single pass. These methods skip the region proposal step and predict the bounding boxes and classes directly in one pass, making it computationally efficient and suitable for real-time detection, especially in resource-limited environments. Though generally less accurate and less effective at detecting small objects, models like You Only Look Once (YOLO) use a fully convolutional neural network (CNN) for fast, single-shot processing. We’ll explore YOLO further in the upcoming section.

**YOLOv11:**

YOLO (You Only Look Once) is an object detection algorithm that uses a single, end-to-end neural network to predict bounding boxes and class probabilities in one pass. Unlike methods like Faster R-CNN that require multiple passes for region proposals and recognition, YOLO detects objects in a single iteration, making it highly efficient. YOLO divides the input image into a grid, where each cell predicts bounding boxes and confidence scores. It also uses techniques like non-maximum suppression to filter overlapping bounding boxes and , improving detection accuracy and speed [2]

YOLOv11 architecture comprises three essential components:

**Backbone:** The backbone of YOLOv11 serves as the primary feature extractor, transforming raw images into multi-scale feature maps through convolutional layers. YOLOv11 introduces the C3k2 block, a computationally efficient version of the Cross Stage Partial (CSP) bottleneck, which uses two smaller convolutions to speed up processing. It retains the Spatial Pyramid Pooling - Fast (SPPF) block from previous versions for enhanced feature extraction and adds a new C2PSA (Cross Stage Partial with Spatial Attention) block. This block improves spatial attention, allowing the model to focus on important image regions, which can enhance detection accuracy for varied object sizes and positions [3].

**Neck:** The neck combines multi-scale features from the backbone and transmits them to the head for final prediction. It uses up sampling and concatenation to enhance multi-scale feature integration. The C3k2 block is also used here to replace the C2f block from previous versions, improving speed and efficiency. Additionally, the C2PSA module introduces spatial attention, helping the model concentrate on key regions, making it particularly effective for detecting small or partially occluded objects [3].

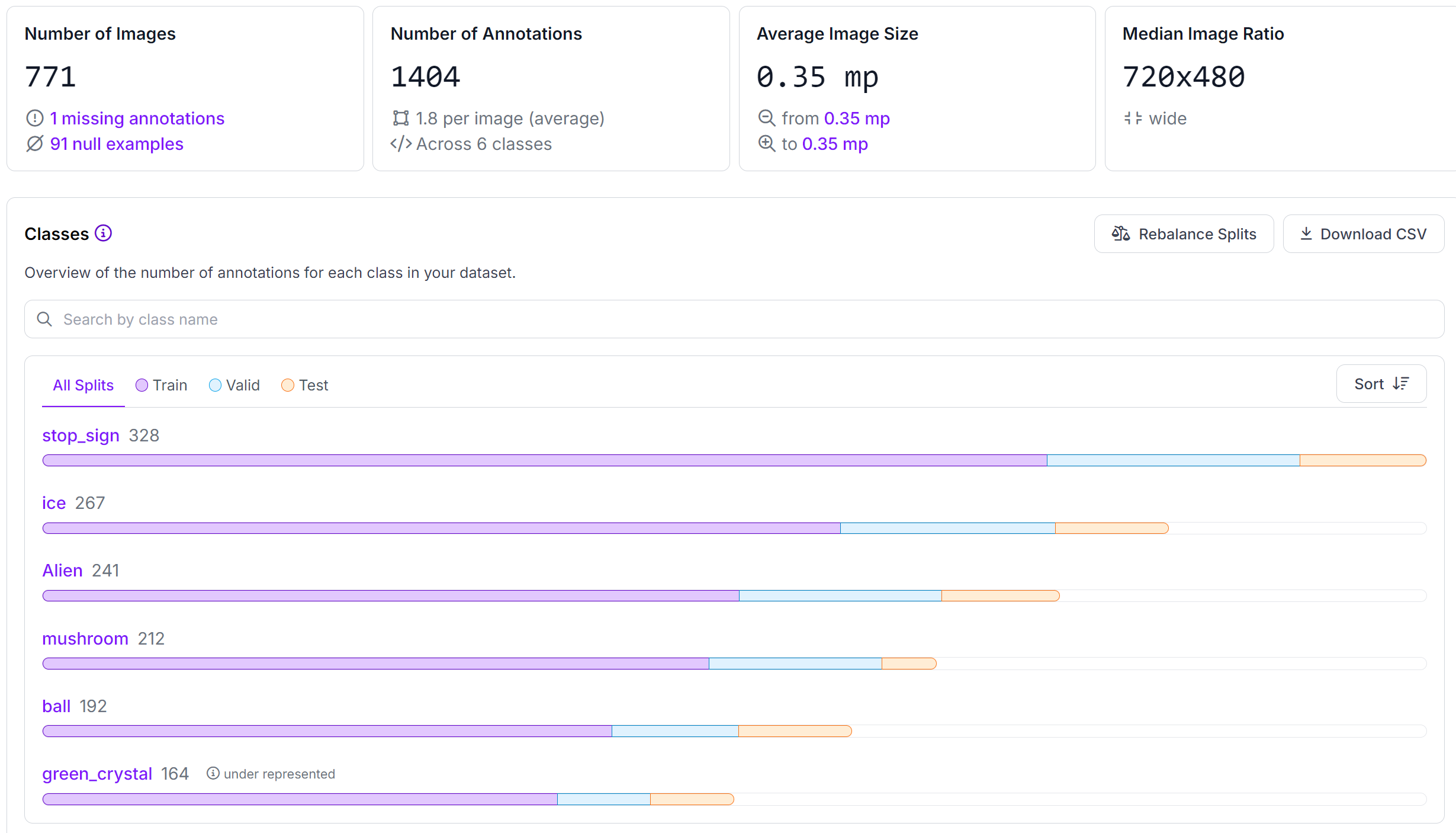
**Head:** The head generates the final predictions for object detection and classification, processing feature maps from the neck. YOLOv11’s head incorporates multiple C3k2 blocks across different layers, enhancing the processing of multi-scale features. This section also uses CBS (Convolution-BatchNorm-SiLU) layers for feature refinement, normalization, and activation, boosting model performance. The final Detect layer outputs bounding box coordinates, objectness scores, and class scores, consolidating the model's detection predictions with improved efficiency and accuracy [3].

**Task 1: Creating dataset of the artifacts**

In task 1 we begin by collecting a dataset of the artifacts which we trained later. I collected the dataset by designing the code to periodically capture images from a ROS camera feed, using a combination of a random walk (provided) and a 2D navigation goal to approach specific artifacts that required capturing.

we initialize a ROS node that subscribes to the camera topic /camera/rgb/image\_raw, receiving images and storing the latest one. An ROS timer is set which triggers the save\_image\_callback function every 1 -2 seconds depending on us to save the most recent image need (I went with saving the image every 2 seconds to remove same artifact redundancy). Using **CvBridge**, the code converts each ROS image message to an OpenCV image format, saving it to a designated folder (/home/vboxuser/Dataset\_images/) with a timestamped filename. This approach efficiently collects a diverse set of images, providing both artifact and non-artifact examples (negative examples) for training a computer vision model.

There are 6 artifacts, and each artifact is repeated thrice in the map totaling to 18 artifacts. Initially I collected around 620 images of various artifacts but soon figured out some are underrepresented in the dataset so went back and collected 230 more images to have approximately equal representation of all the artifacts in the dataset. After cleaning up images we end up with 771 images with 91 negative samples making approximately 11% of the whole dataset. I used Roboflow website to create our dataset.

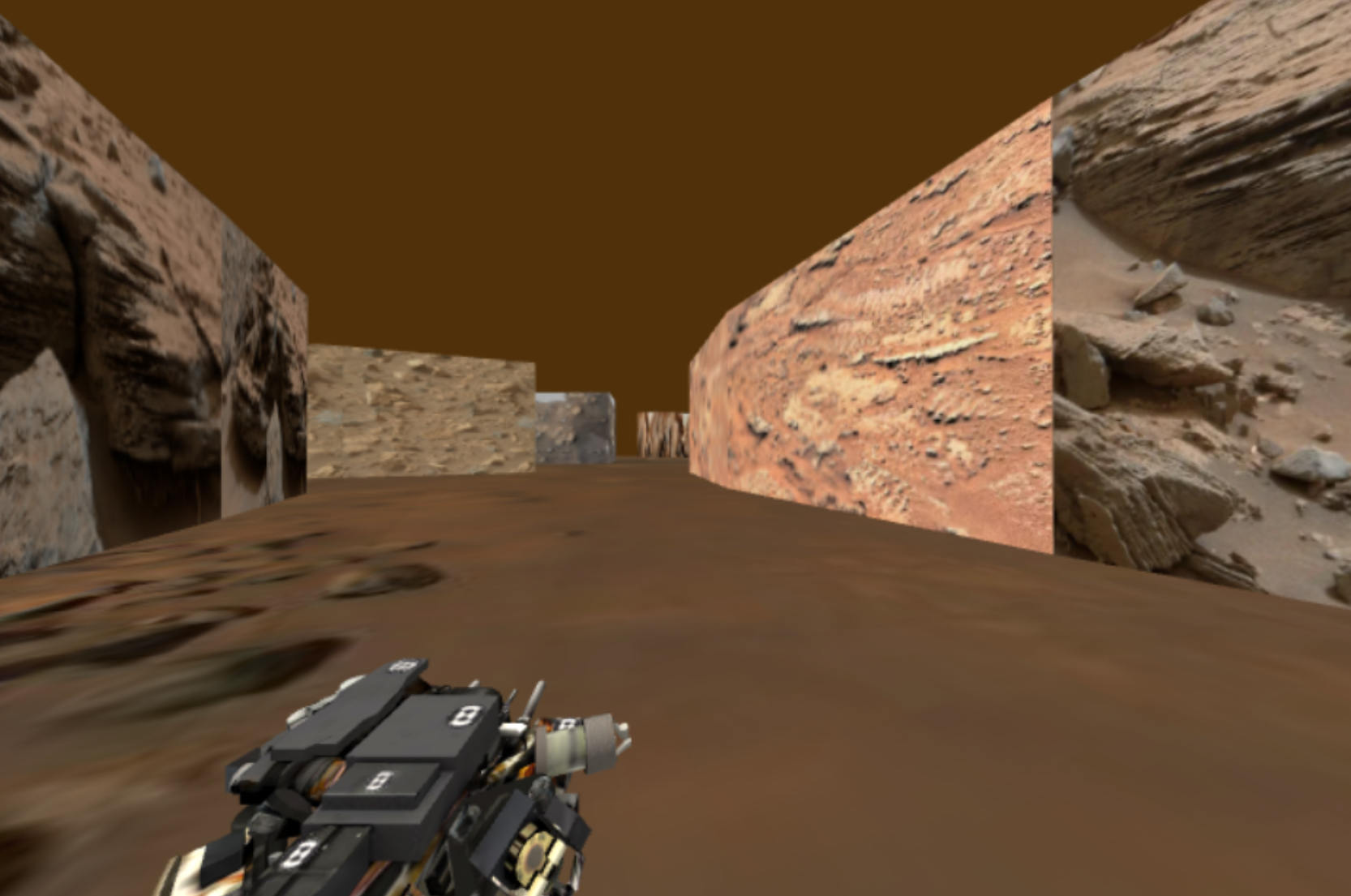


A screenshot of a computer

Description automatically generated

A screenshot of a video game

Description automatically generated

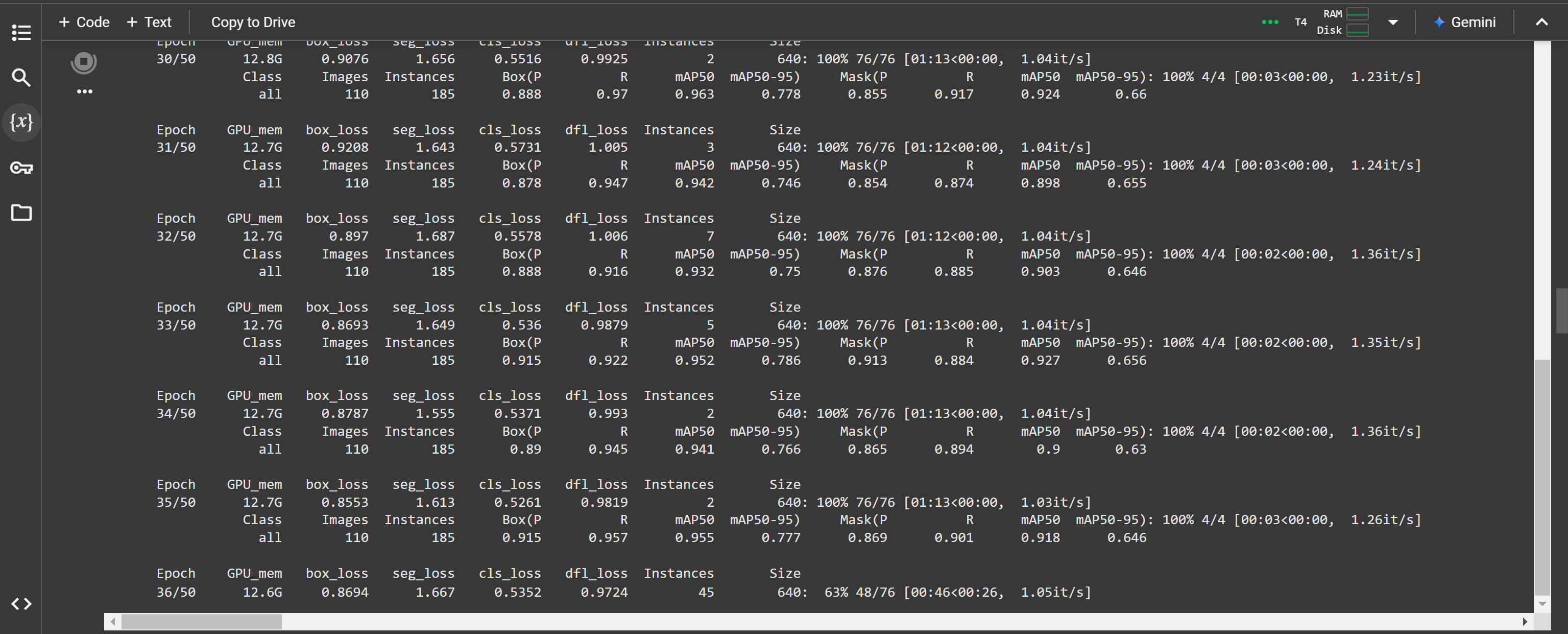
 

*Fig:* *image with artifact (left) and image with no artifact (negative example) (right)*

We have separate classes for each artifact while annotating. I also made use of the Auto augmentations feature in the roboflow to increase our dataset to 1372 images with the split of 86% for training, 9% for validating and 5% for testing. Once we have our dataset we can export it as the yolov11 model to train it.

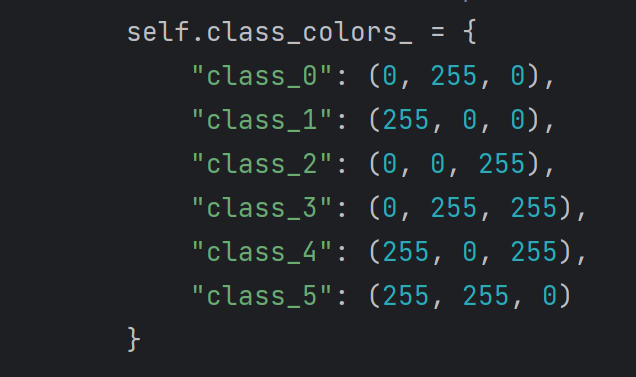
**Task 2: Artifact detection**

Now we have our dataset ready, we need to train the dataset on YOVOv11 model. I used google collab to train it and got great accuracy with less loss when ran with 50 epoch iterations. The results we achieved are shown below. After training, validating, and testing we have the best weight stored in a best.pt file which we download and use in our code to detect the artifact and draw bounding box around them.



First, the YOLO model is loaded from a specified path, with GPU support for efficient processing. The code uses ROS's CvBridge to convert incoming camera images from ROS messages into OpenCV format, making it possible to process them with YOLO.

Once the image is received, it’s resized to the expected input dimensions for the YOLO model. The model then performs inference on the resized image, identifying objects and their bounding boxes, classes, and confidence scores. For each detected object, a bounding box is drawn around it using the coordinates provided by YOLO. The class label and confidence score are also added above the bounding box, with specific colors assigned to up to six classes for clear visual differentiation.



Finally, the processed image, now containing the visualized detections with bounding boxes and labels, is converted back to a ROS-compatible format and published on a dedicated topic.

**Advanced task 1: Artifact localization and display**

I also attempted the advanced task 1 where we need to create a marker at the location of the detected artifact. I only crafted the code to work with one of the artifact in this case ball. The system subscribes to both RGB and depth images from the robot’s camera, leveraging depth information to estimate the 3D position of detected artifacts. Camera intrinsics (focal lengths and principal point coordinates) are found from the camera\_info, which helps in transforming pixel coordinates into camera space. A TF listener provides the robot’s position in the map frame, allowing detected artifacts to be localized in global coordinates. Detected objects are annotated with bounding boxes and labels, with specific colors assigned to different classes, improving visual clarity.

For artifacts identified as “ball,” the code calculates their position relative to the robot, using depth data to obtain the distance at the bounding box center. If the artifact is within a specified distance threshold (2.5 meters) and is either newly detected or significantly repositioned, a marker is created in RViz to represent it. Each marker is configured as a blue sphere, symbolizing the artifact’s location in the global map frame, and published for visualization in RViz. For this I added the marker topic to the rviz.

Still the marker location isn’t accurate to the artifacts location on the map.

**Challenges faced:**

* Initially, I ran YOLO on a virtual machine, but since GPU passthrough was not allowed, inference times were very high (up to 30 seconds per image). I switched to a dual boot setup to utilize the GPU with CUDA installed, reducing inference time to around 20 milliseconds.
* Another issue was that the YOLO model requires input dimensions that are multiples of 32, so I modified the image resolution to 640x448 to meet this requirement.
* While working on advanced task 1, I faced an issue with marker creation: a marker was initially created when an artifact was detected, but it didn’t update with subsequent detections. I modified the code to update the marker’s location when the artifact was detected again. However, this caused the marker to move to the new location instead of creating a new marker and preserving the previous one, leading to less accurate tracking.

**Video Link:**

Below folder link has both Perception 1, 2 & planning 1 combined and advanced task 1 videos.

<https://drive.google.com/drive/folders/1FhbfD-n8Lz7aXk1BougGZfl-bbRVmgnj?usp=sharing>

**References:**

[1] <https://www.v7labs.com/blog/yolo-object-detection>

[2] <https://encord.com/blog/yolo-object-detection-guide/>

[3] K. Jocher, J. Chaurasia, T. Qian, and G. H. Chiu, “YOLOv11: Enhanced single-stage object detection with spatial attention and efficient C3k2 blocks,” *arXiv preprint arXiv:2410.17725*, 2024. [Online]. Available: <https://arxiv.org/abs/2410.17725>

[4] <https://roboflow.com/>

[5] <https://docs.ultralytics.com/modes/train/>

[6] <https://github.com/ultralytics/ultralytics>

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | COMPLETED | TEAM MEMBER | CONTRIBUTION |
| Perception 1 | yes | Gokula Balan Subbiah | Did task 1 fully |
| Perception 2 | yes | Gokula Balan Subbiah | Did task 2 fully |
| Planning 1 | yes | Serey Mongkul Te | Did task1 fully |
| Planning 2 | yes | ⁠Prethivi Raj Elangovan | Did task 2 fully |
| Planning 3 | yes | ⁠Prethivi Raj Elangovan | Did task 3 fully |
| Advanced task 1 | attempted | Gokula Balan Subbiah | Attempted adv task |
| Report writing | yes | everyone | Equal contribution of report writing and video recording |