

Object [Dassault Rafale] Detection on Custom Dataset

Date : 25/11/2023

Prepared by: Santosh Penugurthi

TABLE OF CONTENTS

1: INTRODUCTION.....	5-6
1.1 Background.....	5
1.2 Motivation.....	5
1.3 Objectives	5
1.4 Scope.....	5
1.5 Significance	5
1.6 Structure of the Document	5-6
2: REQUIREMENTS & ANALYSIS	6-8
2.1 Hardware Requirements.....	6
2.2 Software Requirements	6-7
2.3 Functional Requirements.....	7
2.4 Non-Functional Requirements.....	7-8
3: PROPOSED METHOD & WORKFLOW	8-18
3.1 Algorithmic Overview	8-12
3.2 YOLOv8 for Object Detection.....	12-15
3.3 Workflow.....	15-17
3.4 Comparative Analysis.....	17-18
4: IMPLEMENTATION	18-38
4.1 Data Collection.....	18-22
4.2 Data Preparation.....	22-26
4.3 Model Selection & Training.....	26-28
4.4 Model Validation.....	28-33
4.5 Model Testing	34-37
4.6 Model Deployment	37-38
5: CONCLUSION & FUTURE ENHANCEMENTS	38-41
5.1 Conclusion.....	38
5.2 Advantages and Limitations.....	38-39
5.3 Applications.....	39-40
5.4 Challenges Faced and Solutions.....	40-41
5.5 Future Enhancements.....	41
6: REFERENCES.....	42
6.1 References	42

List of Figures

Figure 1: Architecture of CNN	8
Figure 2: A 2D Filter.....	9
Figure 3: Architecture of Convolution layer.....	9
Figure 4: Max Pooling	10
Figure 5: Architecture of YOLOv8.....	13
Figure 6: Workflow	15
Figure 7: Entering YouTube link to collect images in roboflow	19
Figure 8: Processing video to extract images in roboflow	19
Figure 9: Selecting frame rate to collect images in roboflow.....	20
Figure 10: Extracting frames in roboflow	20
Figure 11: Processing collected images in roboflow	21
Figure 12: Uploading collected images in roboflow.....	21
Figure 13: Project overview in roboflow.....	22
Figure 14: Annotating dataset in roboflow	22
Figure 15: Splitting dataset in roboflow	23
Figure 16: Preprocessing dataset in roboflow.....	23
Figure 17: Adding augmentations to dataset in roboflow	24
Figure 18: Generating a version in roboflow.....	24
Figure 19: Dataset version overview in roboflow	25
Figure 20: Exporting dataset in roboflow.....	25
Figure 21: Download code of dataset in roboflow.....	26
Figure 22: Model Training	22
Figure 23: Model Validation	23

Figure 24: Confusion Matrix.....	30
Figure 25: Graph Results	31
Figure 26: Model validation with validation dataset	33
Figure 27: Testing model with test dataset	35
Figure 28: Testing model with unseen image	36
Figure 29: Deployed model overview in roboflow.....	37
Figure 30: Testing deployed model in roboflow.....	38

1 INTRODUCTION

1.1 Background

In response to a practical evaluation of my computer vision skills, I embarked on a task focusing on object detection by Analinear [An imaging solution company]. Given the freedom to choose both the object (any flying object) of interest and the detection model, I selected the renowned Dassault Rafale fighter aircraft as the subject and opted for the powerful YOLOv8 model from Ultralytics. This endeavour is not just a technical task, it is an opportunity to showcase proficiency and creativity in computer vision.

1.2 Motivation

The motivation behind this project is rooted in the desire to demonstrate a comprehensive understanding of object detection, a crucial aspect of computer vision. By selecting Dassault Rafale, a sophisticated and dynamic target, the aim is to showcase the versatility of the chosen model and the applied methodology in addressing real-world challenges.

1.3 Objectives

The overarching objectives of this project are:

- Implementing a robust object detection solution using YOLOv8.
- Showcasing the adaptability of the chosen model in detecting a specific and challenging object like Dassault Rafale.
- Demonstrating proficiency in utilizing cloud-based platforms, particularly Google Colab, for model development.

1.4 Scope

This project's scope extends to:

- Leveraging Roboflow for efficient data collection, annotation, and preparation.
- Employing YOLOv8 in Google Colab to create an accurate and effective object detection model.
- Illustrating the end-to-end process, from data collection to model deployment.

1.5 Significance

The significance of this task lies in its relevance to real-world applications and its potential to showcase skills in data preparation, model development, model validation, model testing and model deployment. The chosen subject Dassault Rafale, adds an element of complexity, providing a robust test case for the implemented object detection solution.

1.6 Structure of the Document

This document unfolds systematically, beginning with an exploration of hardware and software requirements, followed by an in-depth analysis of the proposed method and workflow,

implementation details, and concludes with a reflection on the task's outcomes and potential future enhancements.

The subsequent section delineates the specific hardware and software prerequisites essential for the seamless execution of the task.

2: REQUIREMENTS & ANALYSIS

2.1 Hardware Requirements

The successful implementation of the project necessitates the following hardware components:

- **GPU (Graphics Processing Unit):** A GPU is recommended for accelerated training and inference. While the project can be executed on a CPU, the process will be significantly faster with a GPU. NVIDIA GPUs, such as the GeForce RTX series, are well-suited for deep learning tasks.

Example: NVIDIA GeForce RTX 2080 Ti

- **RAM (Random Access Memory):** Sufficient RAM is crucial, especially when working with large datasets and during model training. A minimum of 16GB is recommended for smooth execution.

Example: 16GB DDR4 RAM

- **Storage Space:** Adequate storage is required for storing datasets, model weights, and project-related files. SSDs are preferred over HDDs for faster data access.

Example: 500GB SSD

- **High-Speed Internet:** A stable and high-speed internet connection is necessary, particularly when working with cloud-based platforms like Google Colab for model training.
- **Webcam:(Optional)** If real-time object detection or live testing is part of the project, having a webcam is beneficial.

Example: Logitech C920 HD Pro Webcam

- **External GPU:(Optional)** For users with laptops lacking a dedicated GPU, an external GPU can be considered to enhance computational capabilities.

Example: Razer Core X eGPU

These hardware requirements are intended to ensure optimal performance and efficiency throughout the project's development and implementation phases. Adjustments may be made based on the specific resources available to the user.

2.2 Software Requirements

To ensure a seamless workflow, the following software components are essential:

- **Google Colab:** Utilized as the primary environment for model development, Google Colab provides free access to GPU resources and facilitates collaborative work on Jupyter notebooks.
- **YOLOv8 (Ultralytics):** The chosen model for object detection. The YOLOv8 implementation by Ultralytics is preferred for its efficiency and ease of use.
- **Roboflow:** Employed for data collection, annotation, and preparation. Roboflow streamlines the process of creating an annotated dataset, a critical component for training a robust object detection model.
- **Python and Dependencies:** Ensure the availability of Python and essential libraries such as NumPy, OpenCV, and TensorFlow. These libraries are crucial for data preprocessing, model configuration, and training.
- **Git:(Optional)** A version control system for tracking changes in the codebase. It is essential for collaborative development.

These software requirements are essential components for developing, training, and deploying the object detection model using YOLOv8. Ensure that the required software is installed and configured appropriately.

2.3 Functional Requirements

The functional requirements delineate the core functionalities that the project must fulfill:

- **Data Collection:** Efficiently gather a diverse set of images of Dassault Rafale from various sources and various light conditions.
- **Data Preparation:** Annotate, split, preprocess, and augment the collected data using Roboflow, ensuring a well-processed and diversified dataset.
- **Model Training:** Implement YOLOv8 on Google Colab, utilizing the annotated dataset to train a robust object detection model.
- **Model Validation:** Assess the model's performance using a validation set to ensure accuracy and generalization.
- **Model Testing:** Evaluate the trained model on a separate test set to validate its real-world applicability.
- **Model Deployment:** If applicable, deploy the trained model to demonstrate its effectiveness in detecting Dassault Rafale.

2.4 Non-Functional Requirements

Beyond the functional aspects, non-functional requirements set the criteria for the project's performance and usability:

- **Computational Efficiency:** Ensure that the implemented solution is computationally efficient, considering both training and inference phases.
- **Accuracy:** Strive for high accuracy in object detection, emphasizing the correct identification of Dassault Rafale instances.

- **Scalability:** Design the solution to scale effectively with an increasing dataset or more complex scenarios.

3: PROPOSED METHOD AND WORKFLOW

3.1 Algorithmic Overview:

Convolutional Neural Networks (CNN)

The proposed method leverages the power of Convolutional Neural Networks (CNNs), a class of deep neural networks particularly adept at image-related tasks. CNNs have proven to be highly effective in capturing intricate patterns and features within images, making them suitable for object detection applications.

Architecture of CNN:

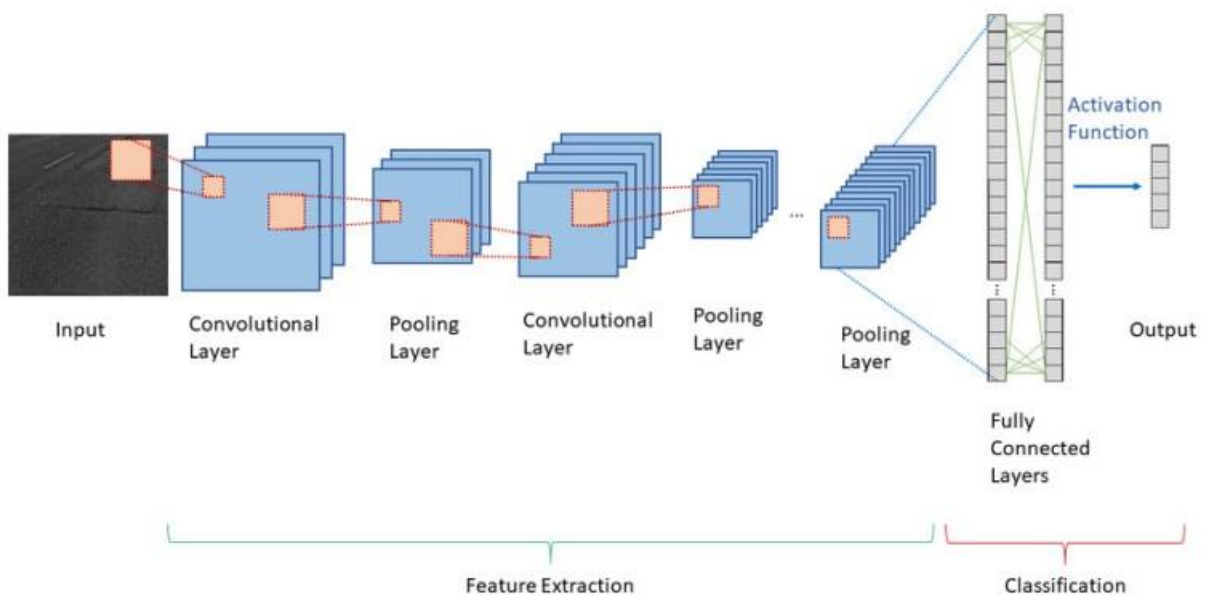


Figure 1. Architecture of CNN

Convolutional Layer

The convolutional layer is considered as the integral building blocks of a CNN. Their primary function is to identify and respond to local patterns or features presented in the previous layer. As shown in the below figure, this task is accomplished by processing a three-dimensional input tensor and creating an output tensor through a set of adaptive filters or kernels. Each filter, in essence, scans over the width and height of the input tensor, performing a dot product computation between the elements of the filter and the input. The outcome is a two-dimensional activation map that reveals how the filter responds at every spatial location. As the network learns, filters are tuned to activate upon recognizing certain visual characteristics such as an edge or a specific colour contrast.

2	0
-1	3

Figure 2. A 2D Image Filter

To better visualize, consider the operation where a filter (depicted in green) of size (2 x 2) aligns with a matching area (represented in orange) within a (4 x 4) input feature map. In the below figure, this process is shown in stages from (a) to (i). Each alignment results in a multiplication operation, the sums of which yield an equivalent data point (shown in blue) on the output feature map. This is iteratively done for each convolution step as the filter slides across the input feature map, the filter takes a step of 1 along the horizontal or vertical position calculating the corresponding output value for the feature map. Intuitively, the network will learn filters that activate when they detect some type of visual feature such as an edge or a particular colour contrast.

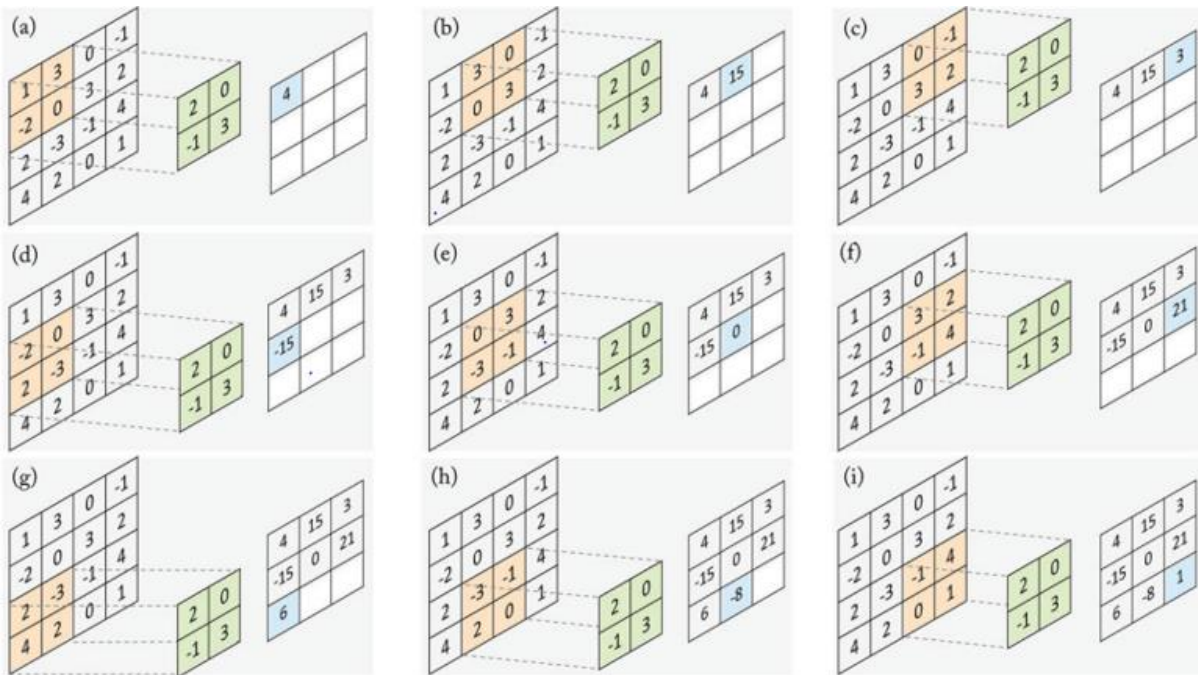


Figure 3. Convolutional Layer Architecture

Pooling Layer

Pooling layers are usually inserted between successive convolutional layers in a CNN architecture. Their function is to progressively reduce the spatial size of the representation, to reduce the number of parameters and computation in the network providing translation invariance and dimensionality reduction, leading to less computational requirements, and preventing overfitting. A pooling layer is an essential component of CNNs. They primarily function to progressively reduce the spatial size (width and height) of the input representation, making the network invariant to minor translations. This down-sampling operation helps in reducing overfitting by providing an abstracted form of the representation, reducing the computational complexity, and hence, the training time. The pooling

layer operates independently on every depth slice of the input and resizes it spatially. A pooling operator operates on individual feature channel, aggregating data of a local region (e.g., a rectangle) and transforming them into one single value. A pooling layer performs operations on sections of the input feature map to merge the feature activations. The method of combination is determined by a pooling function. Like the convolution layer, the size of the pooling area and the stride need to be determined. The max pooling operation, for example, selects the highest activation from a chosen value block, as depicted in the above figure. This window is then moved across the input feature maps in steps determined by the stride size (which is 1 in the below Figure case). Given a pooled region size of ($f \times f$) and a stride (s), the dimensions of the output feature map are thus established. Common choices include max pooling (using the maximum operator) and average pooling (using the average operator), both of which are hand-crafted which are the most common pooling approaches.

$$h' = \left\lfloor \frac{h - f + s}{s} \right\rfloor, w' = \left\lfloor \frac{w - f + s}{s} \right\rfloor. \quad (1)$$

Max Pooling

Max pooling is the most widely used pooling operation and works by defining a spatial neighbourhood usually a 2x2 window and taking the maximum element from the rectified feature map within that window. It can also be understood as a "feature detector" that retains only the highest value in a particular feature map region, discarding all other information as shown in the below figure:

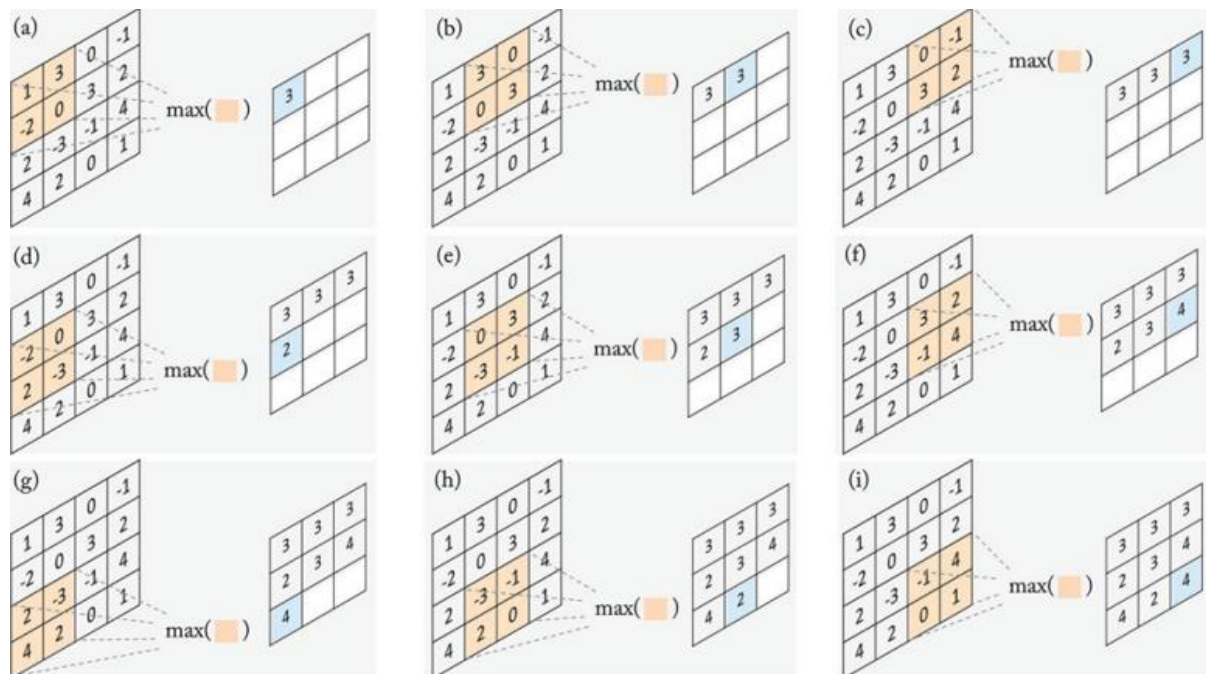


Figure 4. Max Pooling

Average Pooling

Average pooling, as its name suggests, calculates the average of the elements in a feature map region. Unlike max pooling which only keeps the maximum response, average pooling considers the average of the responses, thus retaining more information than max pooling. For instance, in the above figure (a): the maximum element of (1, 3, -2, 0) is (3). On the other hand, the average pooling response will be (0.5). However, average pooling is less common in practice compared to max pooling because it effectively captures the dominant features in a feature map, offering better preservation of critical information. As max pooling provides a form of minor regularization that can mitigate overfitting. Historically, foundational deep learning architectures like LeNet-5 and AlexNet employed max pooling, and their successes cemented its use. Empirically, models using max pooling often outperform those with average pooling on standard benchmarks, further bolstering its popularity in the deep learning community.

Fully Connected Layer

As presented in the architecture image, fully connected layers connect every neuron in one layer to every neuron in another layer. It corresponds essentially to convolution layers with filters of size 1x1. Each unit in a fully connected layer is densely connected to all the units of the previous layer. It is in principle the same as the traditional multi-layer perceptron neural network (MLP). The fully connected layer is usually the last layer in a CNN architecture, and its purpose is to use these features for classifying the input image into various classes based on the training dataset. Its operation can be represented as a simple matrix multiplication followed by adding a vector of bias terms and applying an element-wise nonlinear function.

$$Y = f(W^T x + b) \quad (2)$$

Where \mathbf{x} and \mathbf{Y} are the vectors of input and output activations, respectively, \mathbf{W} denotes the matrix containing the weights of the connections between the layer units and represents the bias term vector.

Key Features of CNNs:

- **Convolutional Layers:** CNNs employ convolutional layers to scan the input image with learnable filters, capturing hierarchical features at different spatial scales.
- **Pooling Layers:** Pooling layers downsample feature maps, reducing spatial dimensions while retaining essential information.
- **Fully Connected Layers:** Fully connected layers at the end of the network help in making predictions based on the learned features.
- **Weight Sharing:** CNNs use weight sharing, enabling the same set of filters to be applied across the entire image, promoting parameter efficiency.

Models Using CNN for Object Detection:

- **YOLO (You Only Look Once):**
 - YOLO employs a CNN architecture to process the entire input image in a single pass.
 - The CNN is responsible for extracting features at different spatial scales, capturing information about objects of various sizes.
 - YOLO divides the input image into a grid and predicts bounding boxes and class probabilities for each grid cell using the features extracted by the CNN.
- **Faster R-CNN (Region-based Convolutional Neural Network):**
 - Faster R-CNN uses a CNN as a backbone (typically pre-trained on ImageNet) to extract hierarchical features from the input image.
 - It introduces a Region Proposal Network (RPN), which is also a small CNN, to propose candidate object regions in the image.
 - The RPN generates region proposals based on the features extracted by the CNN, and these proposals are refined by subsequent layers for final object detection.
- **SSD (Single Shot Multibox Detector):**
 - SSD is a single-shot object detection model that utilizes a series of convolutional layers to predict bounding boxes and class scores.
 - Like YOLO, SSD divides the input image into a grid and predicts bounding boxes and class probabilities for each grid cell using features extracted by the CNN.
 - SSD incorporates feature maps at multiple scales to handle objects of different sizes and aspect ratios in a single pass.

3.2 YOLOv8 for Object Detection

Within the broader scope of CNNs, the YOLOv8 (You Only Look One) algorithm is employed specifically for real-time object detection. YOLOv8 builds upon the principles of CNNs but is tailored for swift and accurate identification of objects within images.

Architecture of YOLOv8:

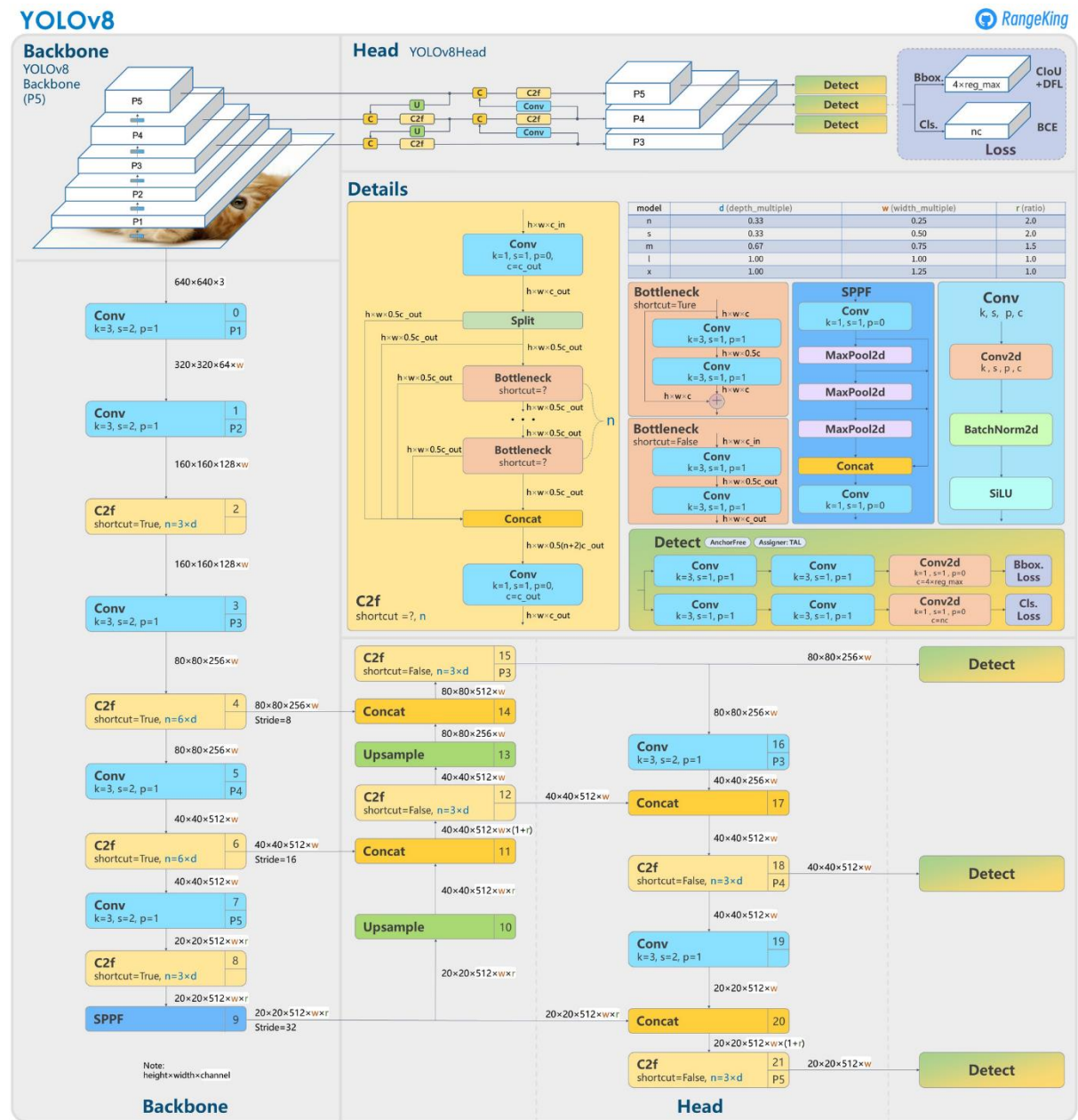


Figure 5. Architecture of YOLOv8

YOLOv8 utilizes a convolutional neural network that can be divided into two main parts: the backbone and the head. A modified version of the CSPDarknet53 architecture forms the backbone of YOLOv8. This architecture consists of 53 convolutional layers and employs cross-stage partial connections to improve information flow between the different layers. The head of YOLOv8 consists of multiple convolutional layers followed by a series of fully connected layers. These layers are responsible for predicting bounding boxes, objectness scores, and class probabilities for the objects detected in an image. One of the key features of YOLOv8 is the use of a self-attention mechanism in the head of the network. This mechanism allows the model to focus on different parts of the image and adjust the importance of different features based on their relevance to the task. Another

important feature of YOLOv8 is its ability to perform multi-scaled object detection. The model utilizes a feature pyramid network to detect objects of different sizes and scales within an image. This feature pyramid network consists of multiple layers that detect objects at different scales, allowing the model to detect large and small objects within an image.

Key Features of YOLOv8:

- **Single Shot Detection:** YOLOv8 follows the single-shot detection paradigm, enabling it to predict bounding boxes and class probabilities in a single forward pass.
- **Anchor Boxes:** The algorithm utilizes anchor boxes to improve localization accuracy, enabling the model to adapt to different object scales.
- **Feature Pyramid Network (FPN):** Incorporating an FPN enhances the model's ability to detect objects at various scales within the input image.
- **Improved Accuracy:** YOLOv8 improves object detection accuracy compared to its predecessors by incorporating new techniques and optimizations.
- **Enhanced Speed:** YOLOv8 achieves faster inference speeds than other object detection models while maintaining high accuracy.
- **Multiple Backbones:** YOLOv8 supports various backbones, such as EfficientNet, ResNet, and CSPDarknet, giving users the flexibility to choose the best model for their specific use case.
- **Adaptive Training:** YOLOv8 uses adaptive training to optimize the learning rate and balance the loss function during training, leading to better model performance.
- **Advanced-Data Augmentation:** YOLOv8 employs advanced data augmentation techniques such as MixUp and CutMix to improve the robustness and generalization of the model.
- **Customizable Architecture:** YOLOv8's architecture is highly customizable, allowing users to easily modify the model's structure and parameters to suit their needs.
- **Pre-Trained Models:** YOLOv8 provides pre-trained models for easy use and transfer learning on various datasets.

The tables of results below come from the Ultralytics report. The detection and segmentation models were trained on the COCO dataset, while classification models are on ImageNet. YOLOv8 provides five models of different sizes - nano (yolov8n), small (yolov8s), medium (yolov8m), large (yolov8l), and extra-large (yolov8x).

Model	size (pixels)	mAP ^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

Table 1. Different models in YOLOv8

I choose medium(yolov8m) model among these as I my task need both accuracy and speed while it is very important of detecting Dassault rafale in real-time. If my task need only accuracy over speed I should go with extra-large(yolov8x) model. If my task need only speed over accuracy I should go with nano(yolov8n) model.

3.3 Workflow:

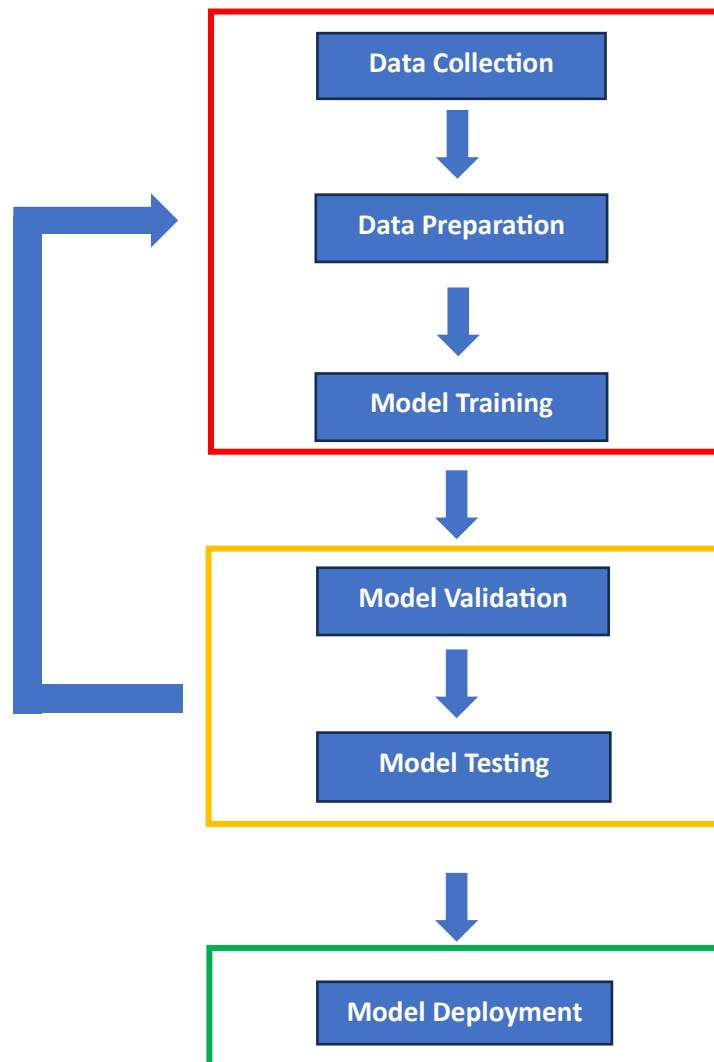


Figure 6. Workflow

3.3.1 Data Collection

- **Collect Best Images and Upload to Roboflow for Dataset Preparation:**
 - Collect Images on the internet featuring the "Dassault Rafale." Collecting images ensures a diverse and comprehensive dataset for training the object detection model.

- **Or loading Images directly from YouTube Video frames using Roboflow:**
 - Collect frames from YouTube videos featuring the "Dassault Rafale." Extracting frames ensures a diverse and comprehensive dataset for training the object detection model.

3.3.2 Data Preparation

- **Annotation using Roboflow:**
 - Utilize Roboflow's annotation tools to label instances of the "Dassault Rafale" in the collected frames. Annotate bounding boxes around the aircraft to create a labelled dataset.
- **Splitting Dataset [Train/Valid/Test]:**
 - Here we split our dataset as Train set, Valid set, Test set.
- **Preprocessing:**
 - Preprocessing allows you to crop, resize, rotate and more before training. Each preprocessing technique has unique benefits. Select each one for a brief rundown of its strengths.
- **Augmentation:**
 - Augmentation performs transforms on your existing images to create new variations and increase the number of images in your dataset. This ultimately makes models more accurate across a broader range of use cases.
- **Export Annotated Dataset:**
 - Export the annotated dataset in the YOLO format compatible with the YOLOv8 model. The exported dataset will include images and corresponding annotation files.

3.3.3 Model Selection & Training

- **Google Colab Setup:**
 - Prepare a Google Colab notebook to take advantage of its free GPU resources. Upload the exported dataset to Google Drive for seamless collaboration and resource utilization.
- **YOLOv8 Configuration:**
 - Download or clone the YOLOv8 repository from the official GitHub page. Configure the YOLOv8 model for the specific dataset, setting parameters such as data paths, model configurations, and training settings.
- **Model Training:**
 - Train the YOLOv8 model using the annotated dataset in Google Colab. Monitor the training process, adjust hyperparameters if needed, and save checkpoints for evaluation.

3.3.4 Model Validation

- **Model Validation:**

- Validate the trained model on the validation set, ensuring it generalizes well to new, unseen data.

3.3.5 Model Testing

- **Model Testing:**

- Test the model on additional images and frames to assess its real-time capabilities and accuracy.

3.3.6 Model Deployment

- **Deployment Consideration:**

- Explore options for model deployment based on project requirements. This may involve using TensorFlow Serving or integrating the model into a web application.

3.4 Comparative Analysis:

Comparative Analysis of selection of YOLO (You Only Look Once) over Faster R-CNN (Region-based Convolutional Neural Network), and SSD (Single Shot Multibox Detector):

Speed and Efficiency:

- **YOLO:** YOLO is known for its speed and efficiency as it processes the entire image in one forward pass, making it well-suited for real-time applications.
- **Faster R-CNN:** Faster R-CNN, while accurate, tends to be slower due to its multi-stage object detection process involving region proposal networks.
- **SSD:** SSD achieves a good balance between speed and accuracy, with a single forward pass for multiple aspect ratios and scales.

Architecture:

- **YOLO:** YOLO uses a single convolutional neural network to predict bounding boxes and class probabilities directly.
- **Faster R-CNN:** Faster R-CNN employs a two-stage architecture, with a region proposal network (RPN) for candidate object regions followed by region-based classification.
- **SSD:** SSD utilizes a single deep neural network to generate predictions at multiple scales.

Accuracy:

- **YOLO:** YOLO provides strong accuracy, especially in detecting large objects, but may face challenges with small object detection.
- **Faster R-CNN:** Faster R-CNN is renowned for its accuracy, particularly in scenarios with small objects, due to its region-based approach.

- **SSD:** SSD offers a good balance between speed and accuracy but may not match the precision of Faster R-CNN in certain cases.

Handling Aspect Ratios:

- **YOLO:** YOLO handles various aspect ratios by dividing the image into a grid and predicting bounding boxes and class probabilities for each grid cell.
- **Faster R-CNN:** Faster R-CNN handles aspect ratios well due to the use of anchor boxes in the RPN.
- **SSD:** SSD addresses aspect ratios by predicting bounding boxes at multiple scales, allowing flexibility in capturing objects of different shapes.

Object Detection for Dassault Rafale:

- **YOLO:** YOLO, with its real-time processing capabilities, could be an excellent choice for detecting Dassault Rafale in images or videos.
- **Faster R-CNN:** Faster R-CNN, with its high precision, might be suitable for accurate detection of Dassault Rafale, particularly in scenarios with small details.
- **SSD:** SSD's balance of speed and accuracy makes it a reliable option for detecting Dassault Rafale in diverse contexts.

	Speed	Accuracy	Ease of implementation
Faster RCNN	Bad	Good	Bad
SSD	Good	Good	Bad
YOLO	Good	Good	Good

Table 2. Comparison among Yolo, Faster R-CNN, SSD

Choosing YOLO for my task is a pragmatic decision, especially considering its real-time processing capabilities, speed, ease of implementation and overall good accuracy. However, the specific requirements of my task and the characteristics of Dassault Rafale influenced me the final choice.

4: IMPLEMENTATION

4.1 Data Collection

4.1.1 Loading Images directly from YouTube Video frames using Roboflow:

Collecting frames from YouTube videos featuring the "Dassault Rafale." Extracting frames ensures a diverse and comprehensive dataset for training the object detection model.

First, I selected a video from YouTube where I found many best frames about Dassault Rafale.

Source: https://www.youtube.com/watch?v=CsVAO_U78Qg&t

Now I opened roboflow workspace and created a new project called “Dassault Rafale Detection Analinear.” There I found upload section and entered my YouTube video link so that it will collect frames. Then I clicked next.

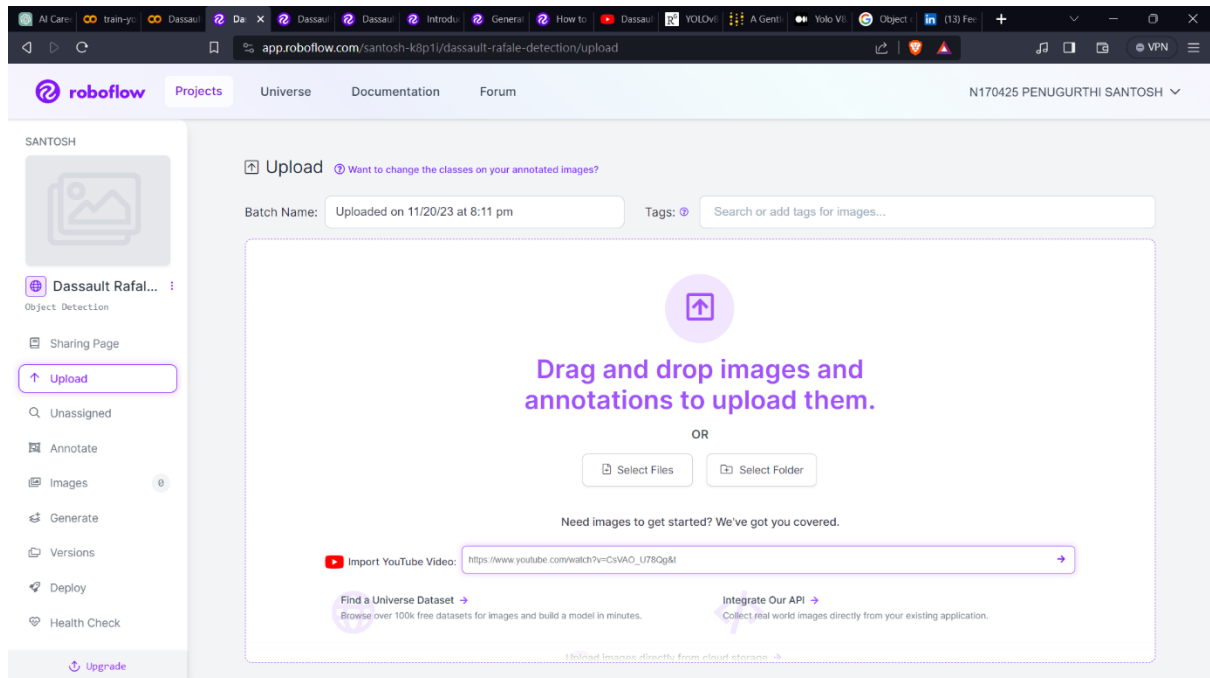


Figure 7. Entering YouTube link to collect images in roboflow

After clicking next it started downloading as shown below.

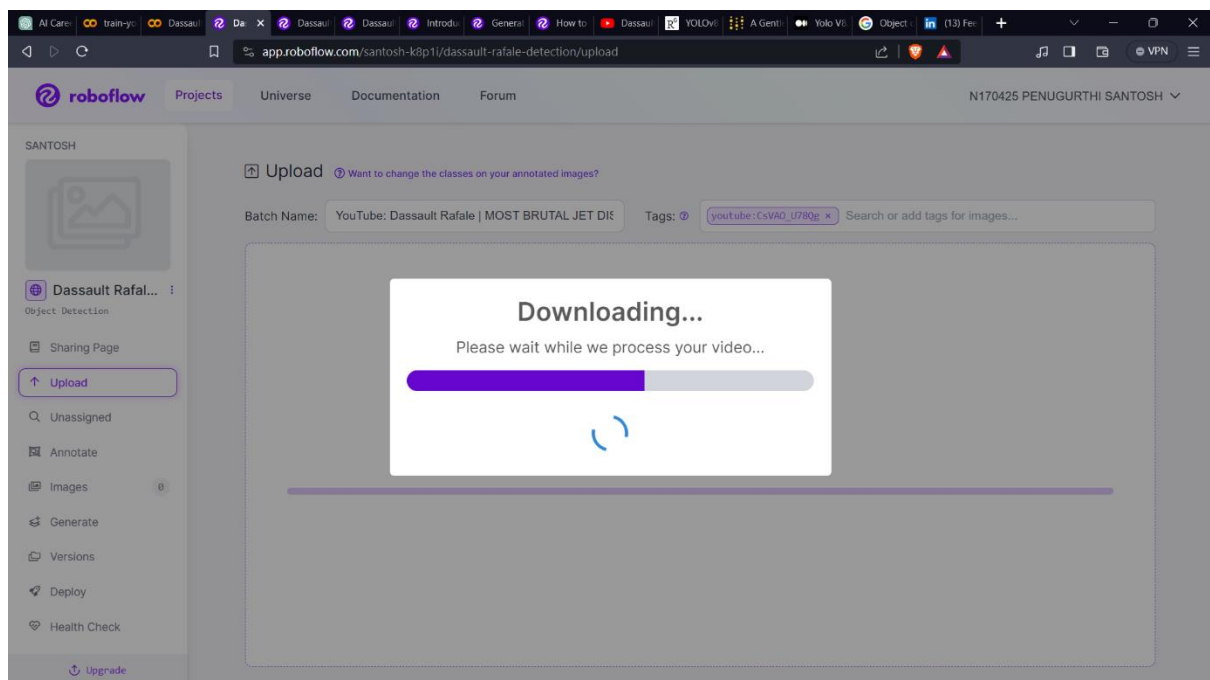


Figure 8. Processing video to extract images in roboflow

Then I got an UI option to select frame rate so that it can collect images.

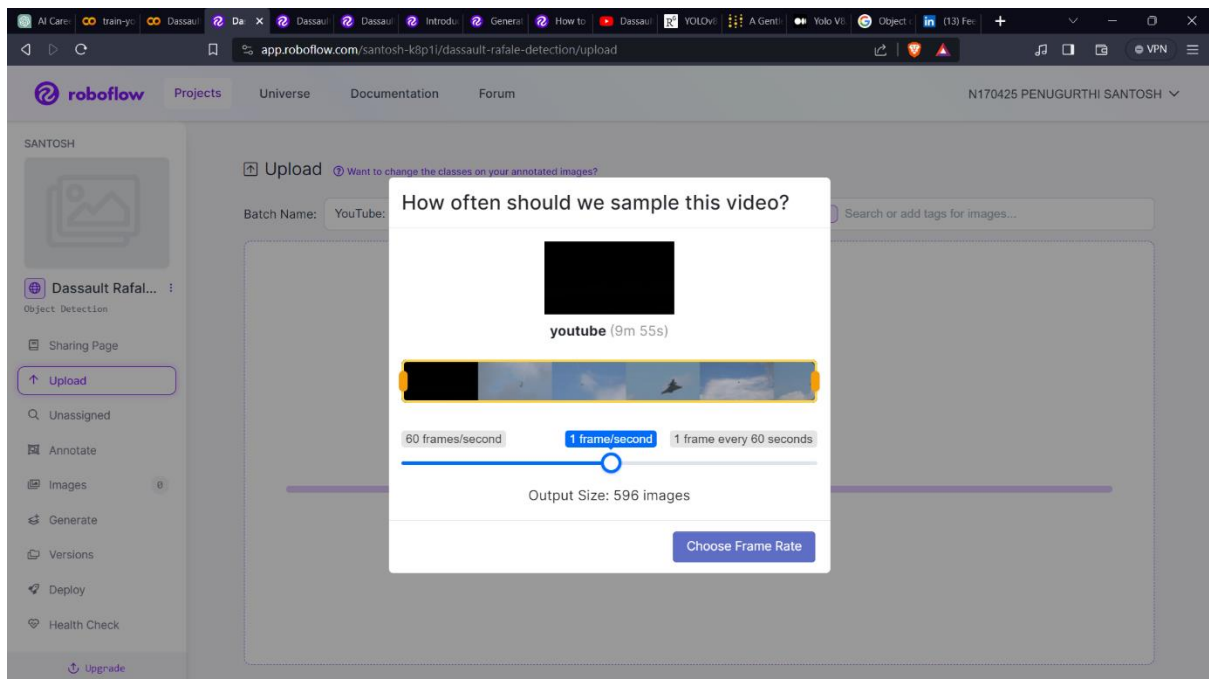


Figure 9. Selecting frame rate to collect images in roboflow

Then it started extracting frames from video.

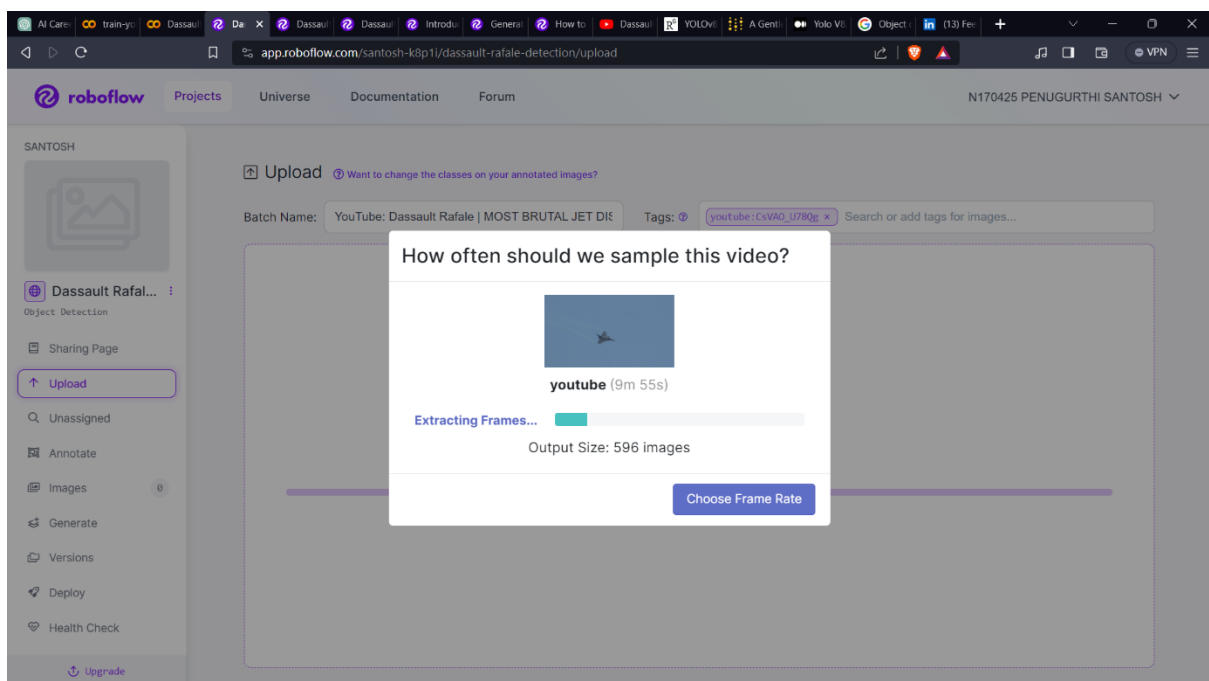


Figure 10. Extracting frames in roboflow

Before going for uploading images to the workspace it will process images for some time.

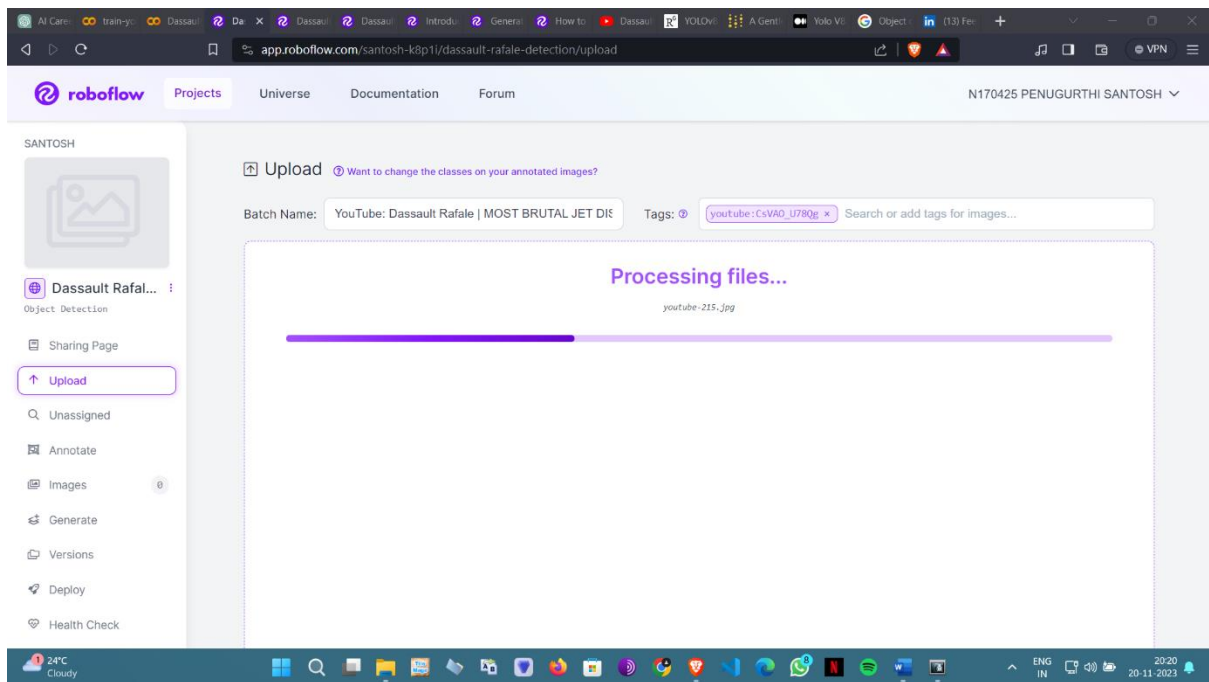


Figure 11. Processing collected images in roboflow

After finishing processing the roboflow will automatically doing uploading task for me.

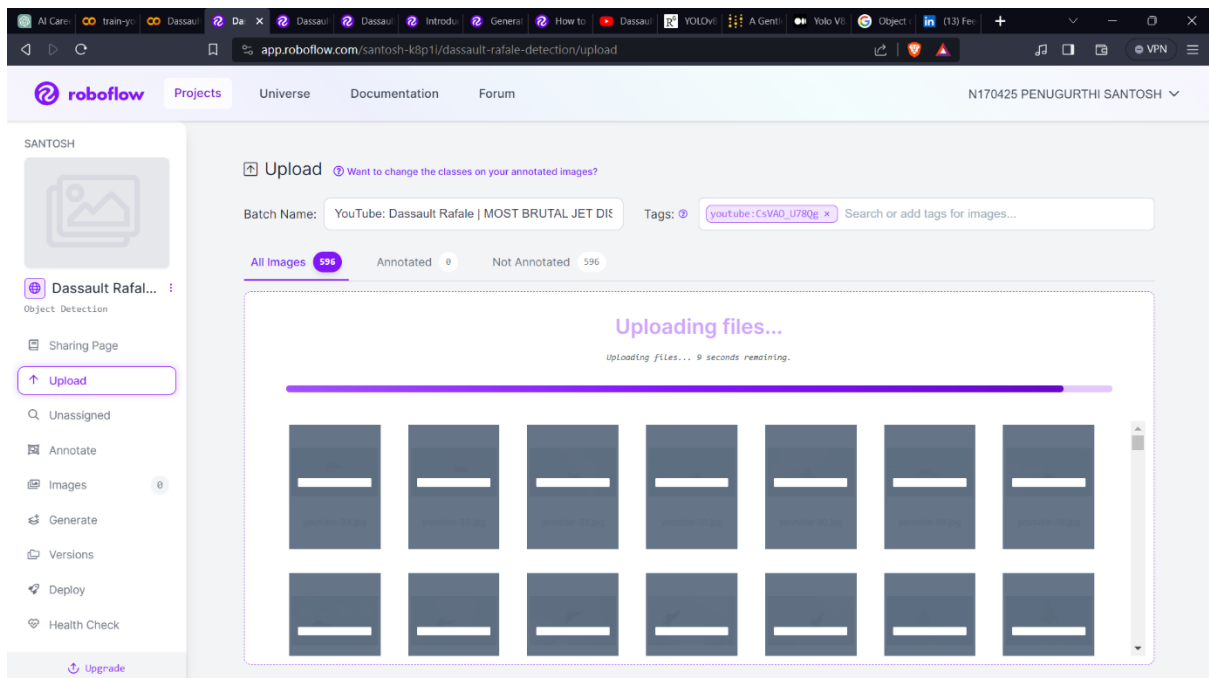


Figure 12. Uploading collected images in roboflow

Now I finished with data collection step. Now I will be directed to data preparation in roboflow itself.

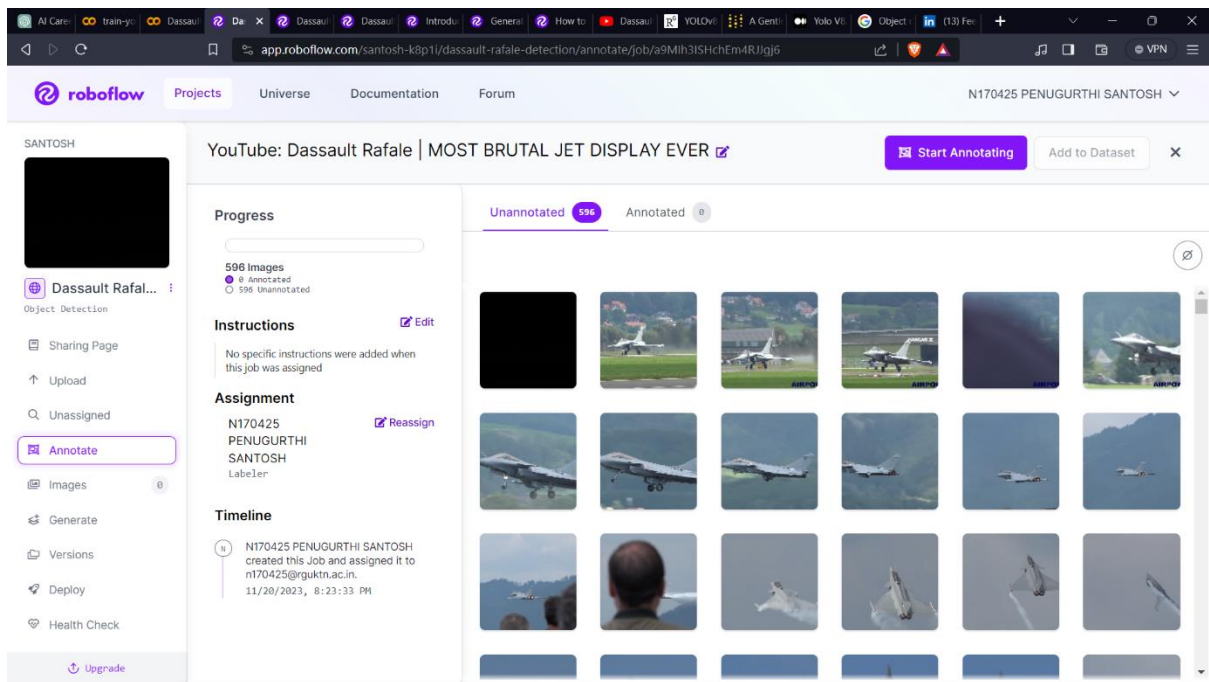


Figure 13. Roboflow project overview

4.2 Data Preparation

4.2.1 Annotation using Roboflow:

Utilizing Roboflow's annotation tools to label instances of the "Dassault Rafale" in the collected frames. Annotating bounding boxes around the aircraft to create a labelled dataset. It took me around 12 to 15 hours.

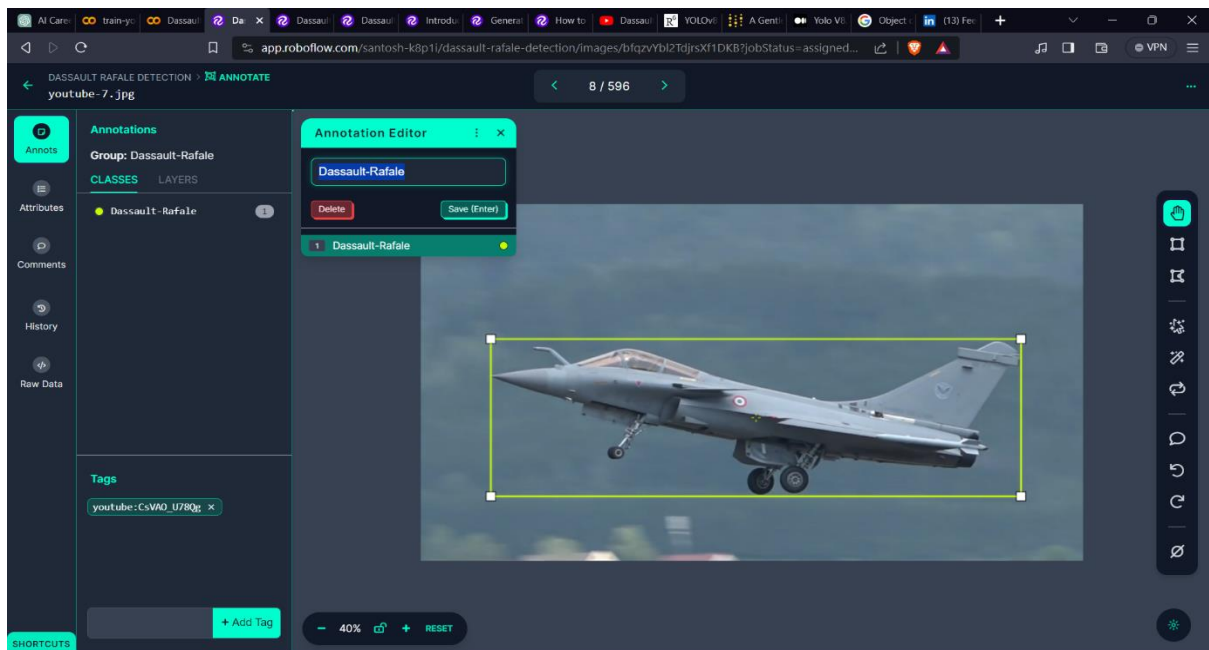


Figure 14. Annotating dataset in roboflow

4.2.2 Splitting Dataset [Train/Valid/Test]:

Here I splitted my dataset as Train set, Valid set, Test set. It is also an UI option we don't need to do much we just need to select the percentage of each set.

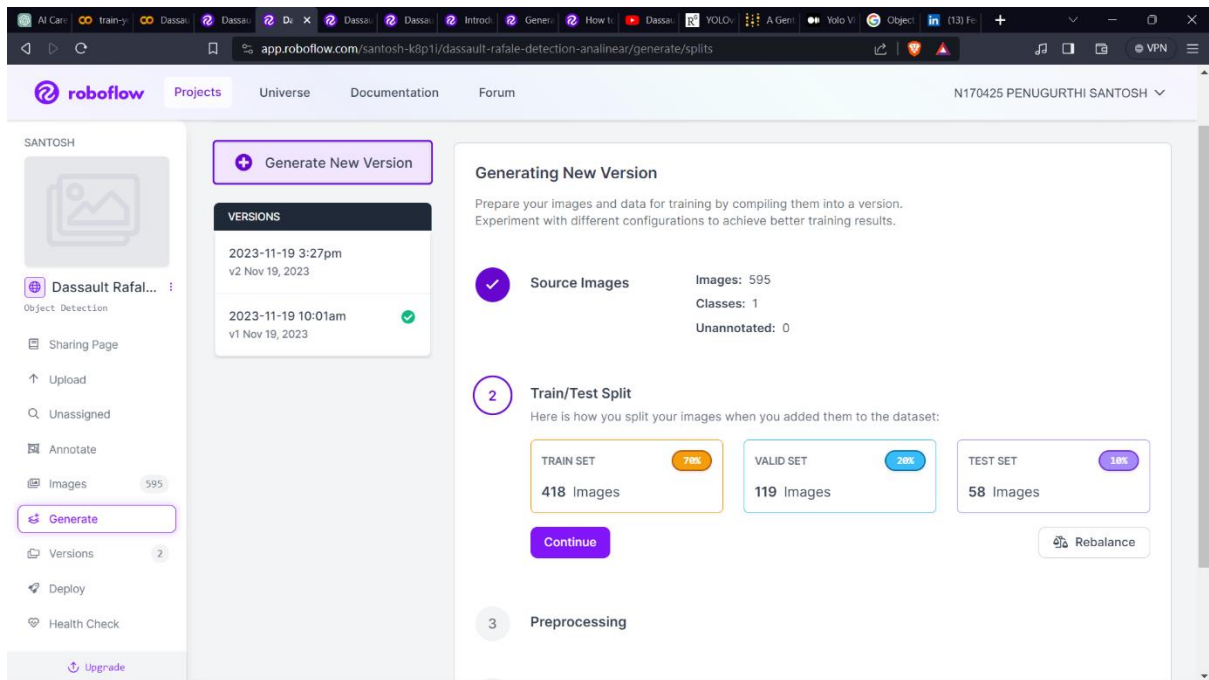


Figure 15. Splitting dataset in roboflow

4.2.3 Preprocessing:

Preprocessing allows us to crop, resize, rotate and more before training. Each preprocessing technique has unique benefits. Here I am going with the default adjustments.

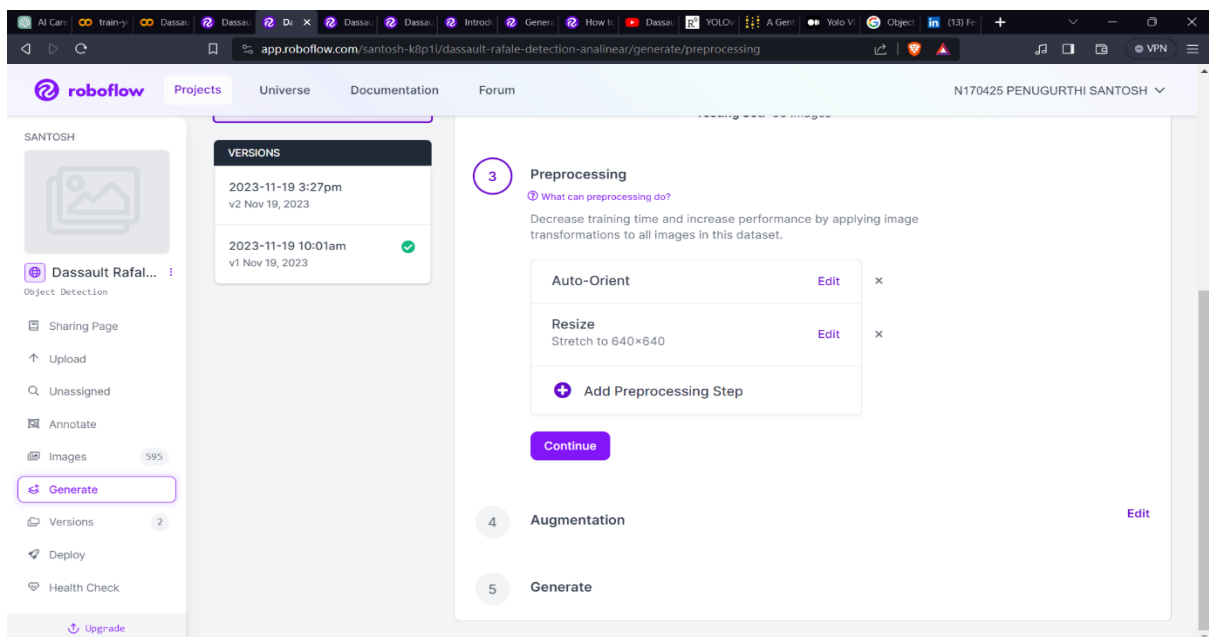


Figure 16. Preprocessing dataset in roboflow

4.2.4 Augmentation:

Augmentation performs transforms on our existing images to create new variations and increase the number of images in your dataset. This ultimately makes models more accurate across a broader range of use cases. Here also I am going with default ones.

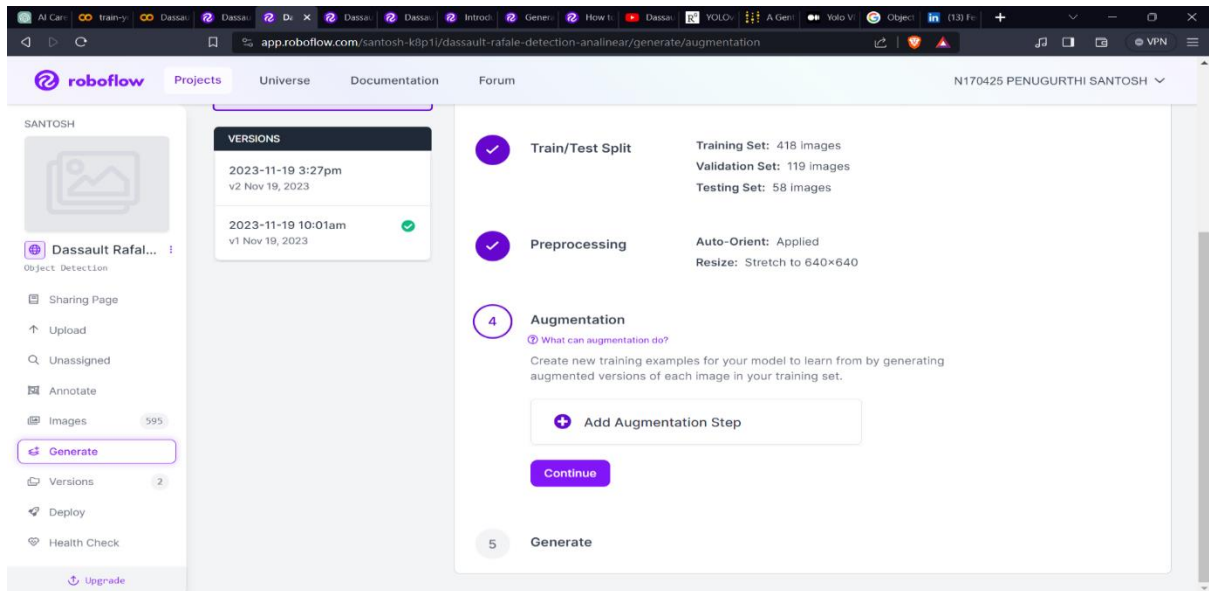


Figure 17. Adding augmentations to dataset in roboflow

4.2.5 Export Annotated Dataset:

Exported the annotated dataset in the YOLO format compatible with the YOLOv8 model. The exported dataset will include images and corresponding annotation files. I clicked on “Generate” button.

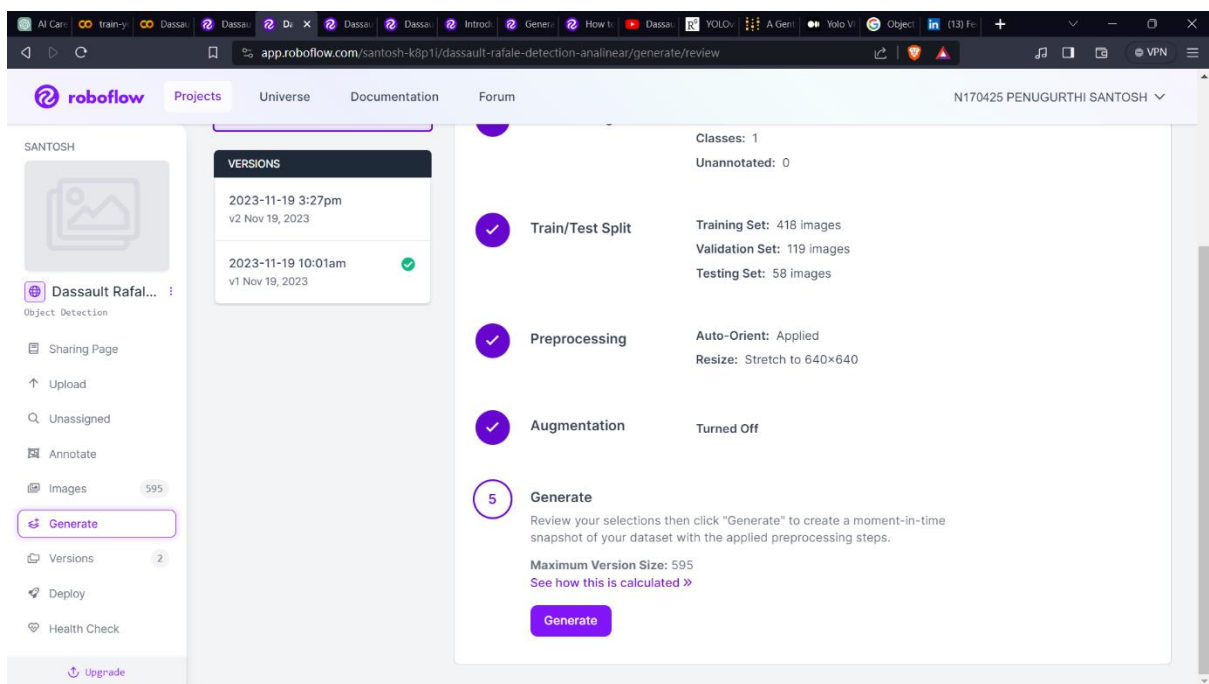


Figure 18. Generating version in roboflow

It created a version of dataset for me.

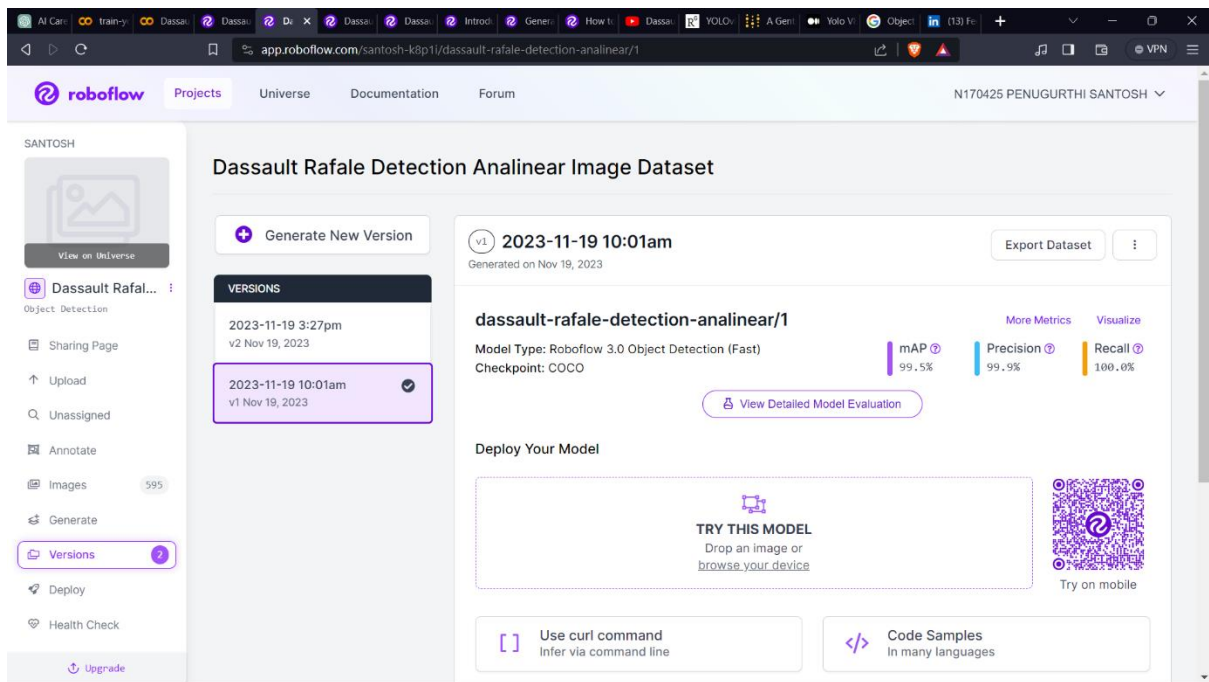


Figure 19. Dataset version overview in roboflow

Now I selected “Export Dataset” option and selected “show download code” to get my dataset for model training.

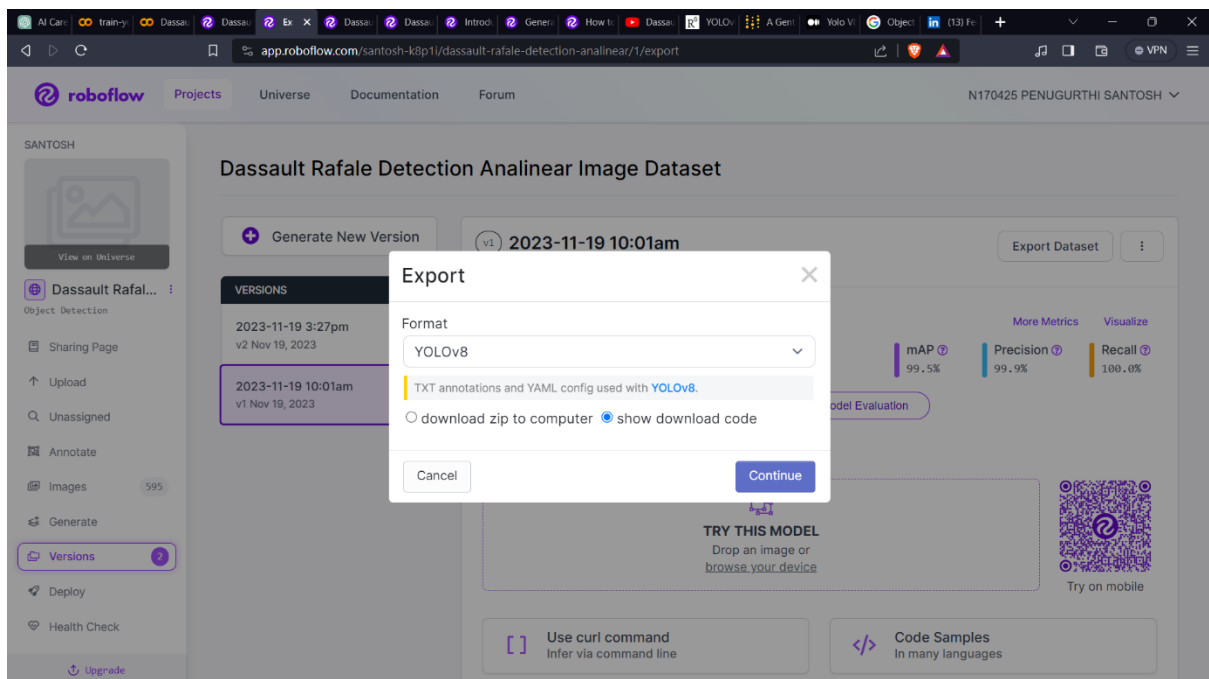


Figure 20. Exporting dataset in roboflow

Here I got some code snipped with some private API key. So that I can use this code in my google colab environment to download the dataset to the working directory.

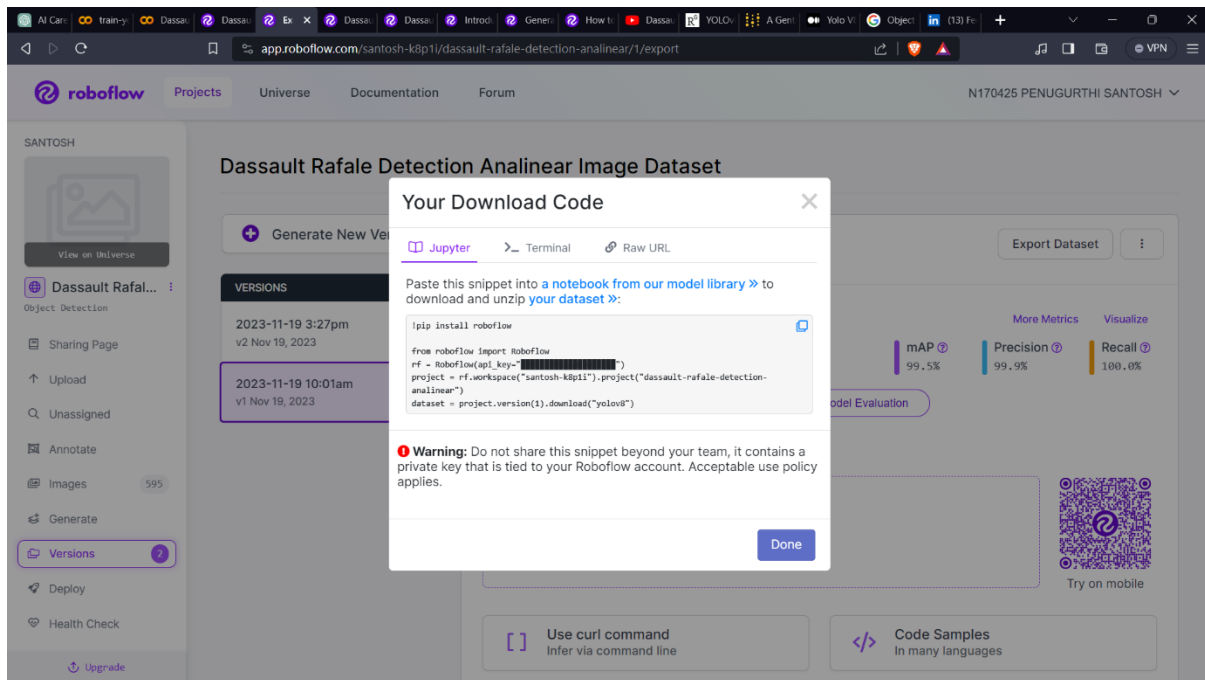


Figure 21. Download code of dataset in roboflow

4.3 Model Selection & Training

4.3.1 Google Colab Setup

A Google Colab notebook is prepared to leverage its GPU resources for efficient model training.

Invidia-smi

nvidia-smi is a powerful tool for GPU management, and it's commonly used in tasks related to deep learning, scientific computing, and graphics processing.

4.3.2 YOLOv8 Configuration

The YOLOv8 model is configured for the specific dataset. Parameters such as data paths, model configurations, and training settings are set to optimize the model for accurate and real-time object detection

```
!pip install ultralytics==8.0.20
from ultralytics import YOLO
```

Above code snippet installs version 8.0.20 of the Ultralytics library using the pip package manager. Following the installation, it imports the YOLO class from Ultralytics. Ultralytics is a computer vision library that offers various functionalities, and in this context, it incorporates an implementation of the YOLO (You Only Look Once) object detection model.

4.3.3 Importing Dataset using Roboflow api

```
!pip install roboflow
from roboflow import Roboflow

rf = Roboflow(api_key="EmHuczrWM84MH6OaBCBf")
project = rf.workspace("santosh-k8p1i").project("dassault-rafale-detection-analinear")
dataset = project.version(6).download("yolov8")
```

I utilized the pip package manager to install the Roboflow Python package, designated by the command "!pip install roboflow." Following the installation, I imported the Roboflow class from the roboflow module and instantiated an object, "rf," of that class, passing my Roboflow API key as a parameter. Subsequently, I accessed my Roboflow workspace named "santosh-k8p1i" and selected the project "dassault-rafale-detection-analinear." Within this project, I specified version 6 and downloaded the dataset labeled "yolov8" using the download method. This code segment allows me to interact with Roboflow programmatically, facilitating the retrieval of datasets for training and evaluation of object detection models in a seamless and reproducible manner.

4.3.4 Model Training

The YOLOv8 model is trained using the annotated dataset. The training process involves optimizing hyperparameters, monitoring loss, and saving checkpoints for subsequent evaluation. Google Colab's GPU resources significantly expedite the training duration.

```
!yolo task=detect mode=train model=yolov8m.pt data={dataset.location}/data.yaml epochs=5 batch=16 imgsz=640
plots=True
```

- **!yolo:** This is a command to interact with the YOLOv8 framework.
- **task=detect:** Specifies that the task is object detection.
- **mode=train:** Indicates that the model should be in training mode.
- **model=yolov8m.pt:** Specifies the initial model weights to start the training. In this case, the model is initialized with the weights from **yolov8m.pt**.
- **data={dataset.location}/data.yaml:** Specifies the path to the data configuration file (in YAML format) required for training. This file likely contains information about the dataset, such as the paths to images, classes, etc.
- **epochs=5:** Sets the number of training epochs to 5. An epoch is one complete pass through the entire training dataset.
- **batch=16:** Sets the batch size to 16. This means that the model will process 16 samples before performing a weight update during each training iteration. The total number of iterations per epoch is determined by the dataset size and batch size.
- **imgsz=640:** Sets the input image size to 640x640 pixels. The model will resize the input images to this size during training.

- **plots=True**: Specifies that training plots (such as loss curves) should be generated and displayed during training.

The number of epochs and batch size interact to influence how the model learns from the data. Training for an appropriate number of epochs helps strike a balance between underfitting and overfitting. Batch size affects the noise in weight updates and computational efficiency. The optimal combination of epochs and batch size depends on the complexity of the dataset and the specific characteristics of the problem.

I experimented with different combinations of epochs and batch sizes, monitored the model's performance on a validation set, and I choose the configuration that provides the best trade-off between training time and generalization. I got some good results by setting epochs equals to 5 and batch as 16.

Figure 22. Model Training

By performing training mode we can observe the things in below figure. There we can compare the training process with yolov8 architecture. We can observe how yolov8 works in real time. It is training with 400 images and 0 backgrounds.

4.4 Model Validation

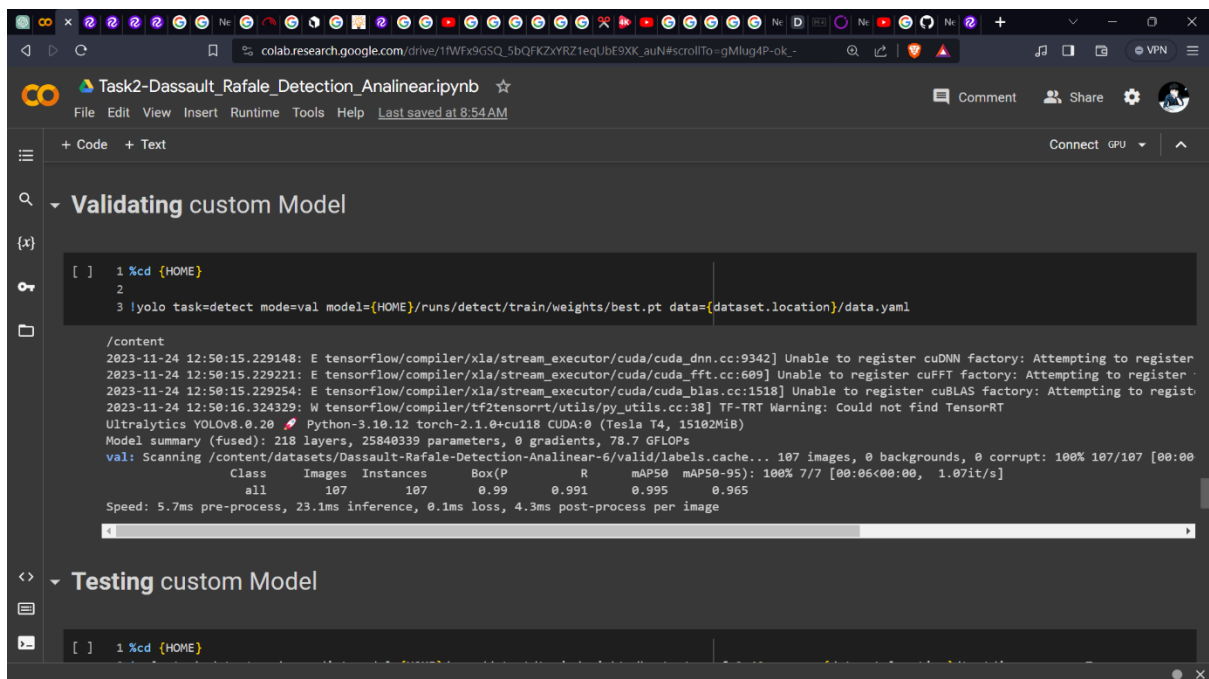
4.4.1 Model Validation

- The trained YOLOv8 model undergoes validation using the designated validation set. This step ensures that the model performs effectively and accurately on data it has not encountered during training.

```
!yolo task=detect mode=val model={HOME}/runs/detect/train/weights/best.pt data={dataset.location}/data.yaml
```

- **!yolo:** This is a command-line utility, likely provided by the YOLO framework or a related tool.
- **task=detect:** Specifies that the task at hand is object detection.
- **mode=val:** Indicates that the model is in validation (evaluation) mode, where it assesses its performance on a separate validation dataset.
- **model={HOME}/runs/detect/train/weights/best.pt:** Specifies the path to the YOLO model checkpoint file used for validation. In this case, it's pointing to the "best.pt" file located in the specified directory.
- **data={dataset.location}/data.yaml:** Points to the configuration file (in YAML format) that provides information about the dataset, including the paths to the training and validation data, number of classes, etc. The {dataset.location} placeholder likely refers to the location of your dataset.

Overall, I used this command to evaluate the performance of the YOLO model stored in "best.pt" on the validation dataset using the configuration specified in the "data.yaml" file. It helps to assess how well the model generalizes to unseen data and provides metrics such as precision, recall, and mean average precision (mAP).



```
[ ] 1 %cd {HOME}
2
3 !yolo task=detect mode=val model={HOME}/runs/detect/train/weights/best.pt data={dataset.location}/data.yaml

/content
2023-11-24 12:50:15.229148: E tensorflow/compiler/xla/stream_executor/cuda/cuda_dnn.cc:9342] Unable to register cuDNN factory: Attempting to register
2023-11-24 12:50:15.229221: E tensorflow/compiler/xla/stream_executor/cuda/cuda_fft.cc:609] Unable to register cuFFT factory: Attempting to register
2023-11-24 12:50:15.229254: E tensorflow/compiler/xla/stream_executor/cuda/cuda_blas.cc:1518] Unable to register cuBLAS factory: Attempting to regist
2023-11-24 12:50:16.324329: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not find TensorRT
Ultralytics YOLOv8.0.20 Python-3.10.12 torch-2.1.0+cu118 CUDA:0 (Tesla T4, 15102MiB)
Model summary (fused): 218 layers, 25840339 parameters, 0 gradients, 78.7 GFLOPs
val: Scanning /content/datasets/Dassault-Rafale-Detection-Analinear-6/valid/labels.cache... 107 images, 0 backgrounds, 0 corrupt: 100% 107/107 [00:00:00.00]
      Class  Images  Instances  Box(P)      R    mAP50  mAP50-95)  100% 7/7 [00:06<00:00, 1.07it/s]
      all      107      107      0.99    0.991    0.995    0.965

Speed: 5.7ms pre-process, 23.1ms inference, 0.1ms loss, 4.3ms post-process per image
```

Figure 23. Model Validation

In evaluating my YOLO object detection model, I achieved promising results with a precision of 0.99, indicating a high accuracy in correctly identifying positive detections among the predicted ones. The recall of 0.991 is also impressive, suggesting that the model effectively captured a significant portion of the actual positive instances in the dataset. Furthermore, the mAP50 score of 0.995 and mAP50-

95 of 0.965 are indicative of the model's robustness in handling various object scales and its ability to maintain high precision across a range of confidence thresholds. These metrics collectively demonstrate the effectiveness of the trained model in accurately detecting and localizing objects within the specified dataset.

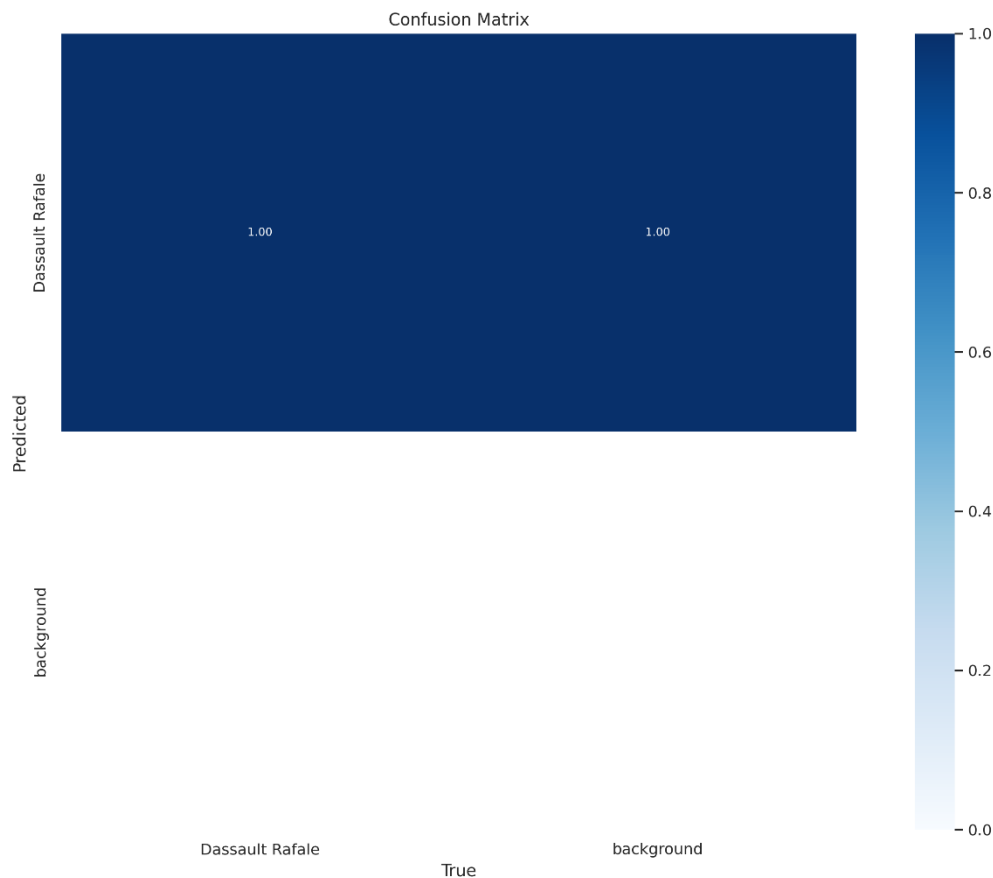


Figure 24. Confusion matrix

Results I got in the Confusion Matrix:

- True Positive (TP) : 1.00
- True Negative (TN) : 0.00
- False Positive (FP) : 1.00
- False Negative (FN) : 0.00

Analysis based on results:

- **True Positive (TP)** : The model correctly identified positive instances.
- **True Negative (TN)** : There are no actual background instances in my dataset, so TN is not applicable or meaningful in this context.
- **False Positive (FP)** : Column normalized.
- **False Negative (FN)** : The model correctly identified zero negative instances.

The “normalized” term means that each of these groupings is represented as having 1.00 samples. Thus, the sum of each column in a balanced and normalized confusion matrix is 1.00, because each column sum represents 100% of the elements in a particular topic, cluster, or class.

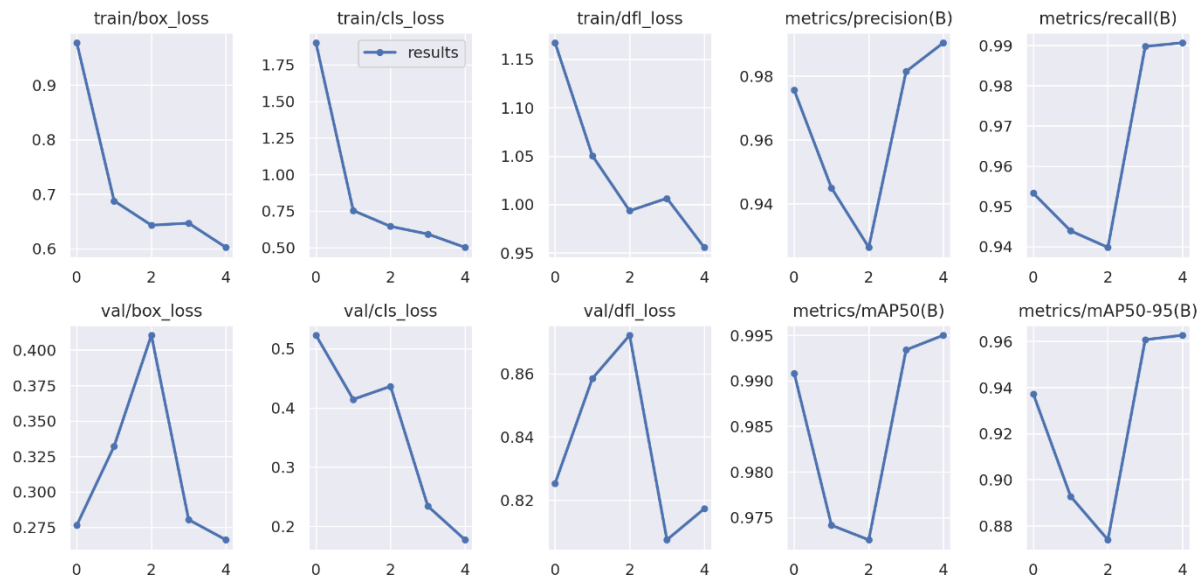


Figure 25. Graph Results

The relationship between the number of training epochs and various metrics, including box loss, class loss, and dfl loss, as well as precision, recall, and other performance metrics, can provide insights into how the model is learning and improving over time during the training process. Here's a general understanding of these relationships:

- **Box Loss, Class Loss, and DFL Loss:**

- These are commonly used loss functions that measure the difference between the predicted values and the ground truth.
- In the early epochs, loss values are often higher as the model is still learning to capture patterns in the data.
- As training progresses, loss values generally decrease, indicating improved model performance.

- **Precision and Recall:**

- Precision and recall are performance metrics commonly used in object detection tasks.
- Precision is the ratio of true positive predictions to the total number of positive predictions (precision = $TP / (TP + FP)$).

- Recall is the ratio of true positive predictions to the total number of actual positive instances ($\text{recall} = \text{TP} / (\text{TP} + \text{FN})$).
- The precision-recall trade-off can vary during training epochs. Initially, one may improve at the expense of the other, and vice versa.
- **Graphs and Visualizations:**
 - Plotting precision and recall over epochs can provide insights into the model's ability to make accurate positive predictions while minimizing false positives and false negatives.
 - These graphs help in understanding how well the model is balancing precision and recall at different stages of training.
- **Epochs and Model Convergence:**
 - With more epochs, the model tends to converge, meaning it learns the underlying patterns in the data more effectively.
 - However, increasing the number of epochs too much might lead to overfitting, where the model becomes too specific to the training data and performs poorly on new data.
- **Adjusting Learning Rates:**
 - Sometimes, adjusting the learning rate during training can impact how quickly the model converges and the stability of the training process.

It is crucial to monitor these metrics over epochs and potentially adjust training strategies (learning rates, data augmentation, etc.) based on the observed trends. Regular evaluation on a validation set can help in selecting the best model checkpoint based on performance metrics.

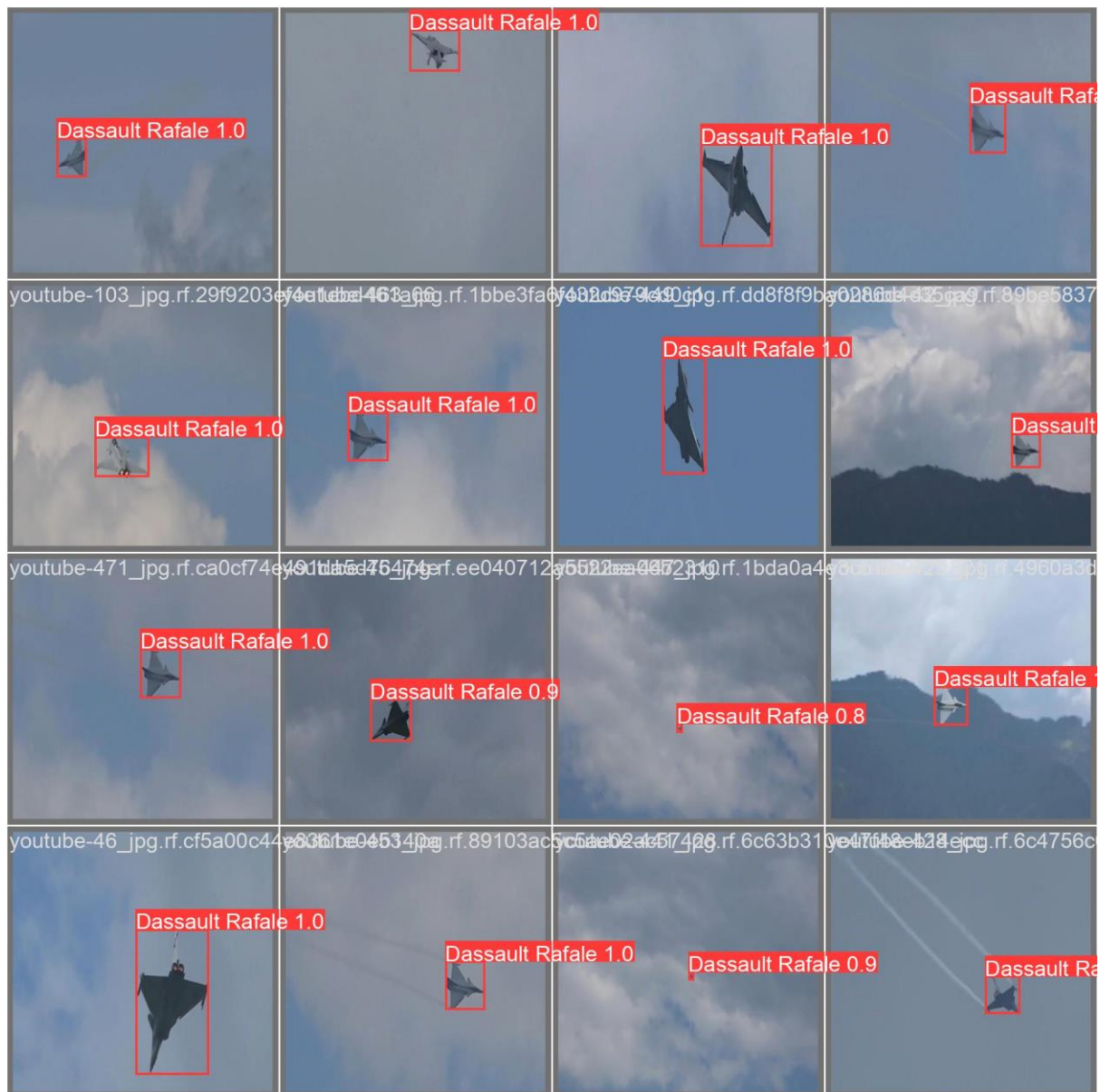


Figure 26. Model Validation with validation set

Having successfully completed the validation phase and achieving commendable results with the best.pt model, I am now proceeding to the testing phase using unseen data. This crucial step involves evaluating the model's performance on data it has not encountered during training or validation. By employing unseen images and possibly videos, I aim to assess the generalization capability of the model and ensure its robustness in real-world scenarios. This testing phase serves as a critical checkpoint to validate the model's ability to accurately detect and classify objects, particularly focusing on instances not present in the training or validation sets. The insights gained from this testing process will contribute to a comprehensive understanding of the model's practical applicability and reliability.

4.5 Model Testing

4.5.1 Model Testing with prepared test dataset:

The model is tested on additional images and frames beyond the training and validation sets. This real-world testing evaluates the model's real-time capabilities and its accuracy in diverse scenarios.

I ensured that my test dataset is diverse and representative of the scenarios my model might encounter in real-world applications.

I analysed any specific patterns or challenges that my model faced in the prepared test dataset.

```
!yolo task=detect mode=predict model={HOME}/runs/detect/train/weights/best.pt conf=0.40  
source={dataset.location}/test/images save=True
```

- **!yolo**: Initiates the YOLO command.
- **task=detect**: Specifies the task as object detection.
- **mode=predict**: Sets the mode to prediction, indicating that the model will be used for making predictions.
- **model={HOME}/runs/detect/train/weights/best.pt**: Specifies the path to the YOLOv8 model file (best.pt) that will be used for predictions. The model is in the specified directory.
- **conf=0.40**: Sets the confidence threshold for predictions. Only predictions with a confidence score greater than or equal to 0.40 will be considered valid.
- **source={dataset.location}/test/images**: Specifies the source directory containing the test images on which predictions will be made. The test images are in the specified directory.
- **save=True**: Indicates that the results of the predictions will be saved. This is useful for reviewing and analyzing the model's performance on the specified test images.

This command is designed for the prediction phase, where the YOLOv8 model is applied to test data, and the results are saved for further evaluation. Below we can see the result image. There we can see it has successfully detected Dassault Rafale with less box loss and class loss.



Figure 27. Testing model with test dataset

4.5.2 Model Testing with unseen images:

I selected images manually that are relevant to my application and may not be present in the training or test dataset. I evaluated how well my model performed on these images, paying attention to any potential issues or unexpected behaviour.

```
from ultralytics import YOLO
from PIL import Image

# Load a pretrained YOLOv8n model
model = YOLO('./Best_Trained_Model_with_Custom_Dataset/best.pt')

# Define path to the image file
source = './Input/task2-test.jpg'

# Run inference on the source
results = model(source) # list of Results objects

# Show the results
for r in results:
    im_array = r.plot() # plot a BGR numpy array of predictions
    im = Image.fromarray(im_array[..., ::-1]) # RGB PIL image
    im.show() # show image
    im.save('./Output/task2-result_.jpg') # save image
```

Below we can see the result image. There we can see it has successfully detected Dassault Rafale with less box loss and class loss.



Figure 28. Testing model with unseen image

4.5.3 Model Testing with video tracking:

Video tracking added a temporal dimension to my evaluation, assessing how well my model could maintain object tracks over time. I considered using metrics related to tracking accuracy, like Intersection over Union (IoU) across frames. I analysed the robustness of the tracking algorithm in different scenarios, such as occlusions, object scale changes, or rapid movements.

I used best.pt model for my video tracking and I have generated two output videos using this model. One is object detection and another one is object detection along with movement patterns. I have saved these videos in the Output Directory.

The output video1: ./Output/task2-output-video1.mp4

The output video2: ./Output/task2-output-video2.mp4

In addition to quantitative metrics, I considered qualitative analysis by visually inspecting the results. I looked for instances where my model succeeded or failed, and I tried to understand the reasons behind its behaviour.

I made sure to document my testing methodology, including the datasets used, evaluation metrics, and any observations or insights gained from the testing process. This information will be valuable when presenting or sharing my model evaluation results.

4.6 Model Deployment (Optional)

4.6.1 Deployment Consideration

If real-time object detection is a project requirement, deployment options are explored. TensorFlow Serving or integration into a web application are considered for making the model accessible beyond the training environment.

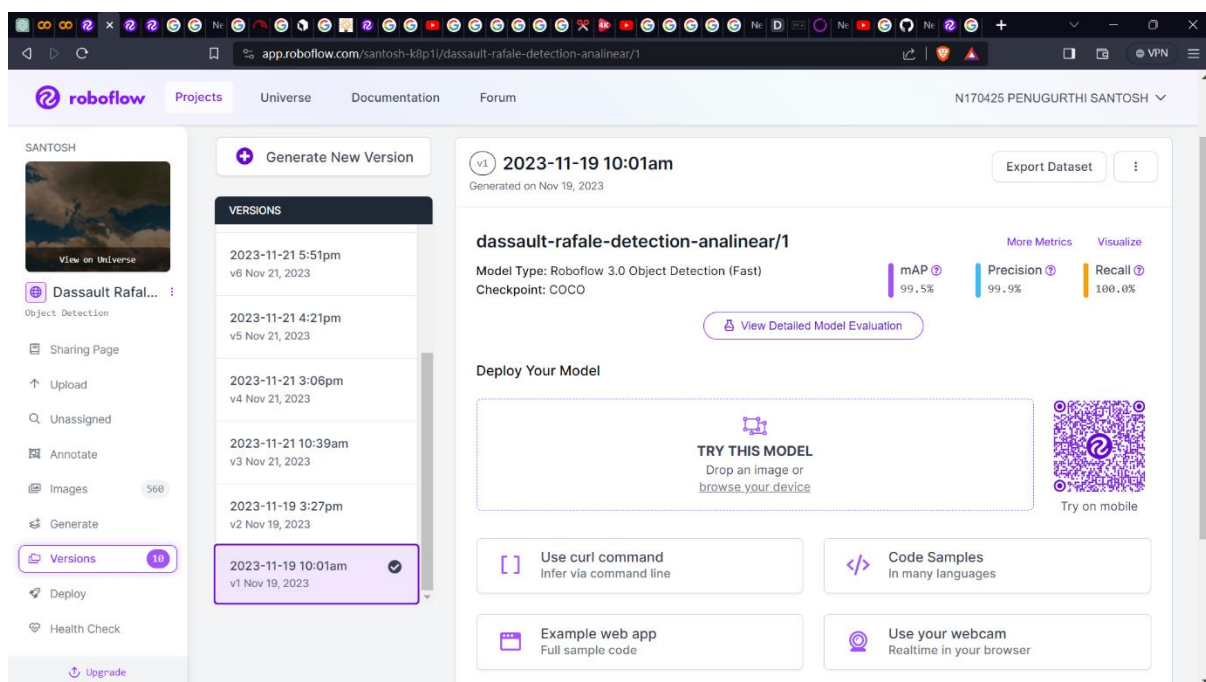


Figure 29. Deployed Model Overview in roboflow

During the model deployment phase, I deployed my YOLOv8 model with dataset version 1. Unfortunately, the results obtained were not up to the expected standards, raising concerns about the effectiveness of the deployment. Notably, the attempt to deploy another best model trained with dataset version 6 required a premium subscription, indicating potential cost implications beyond the free credits available.

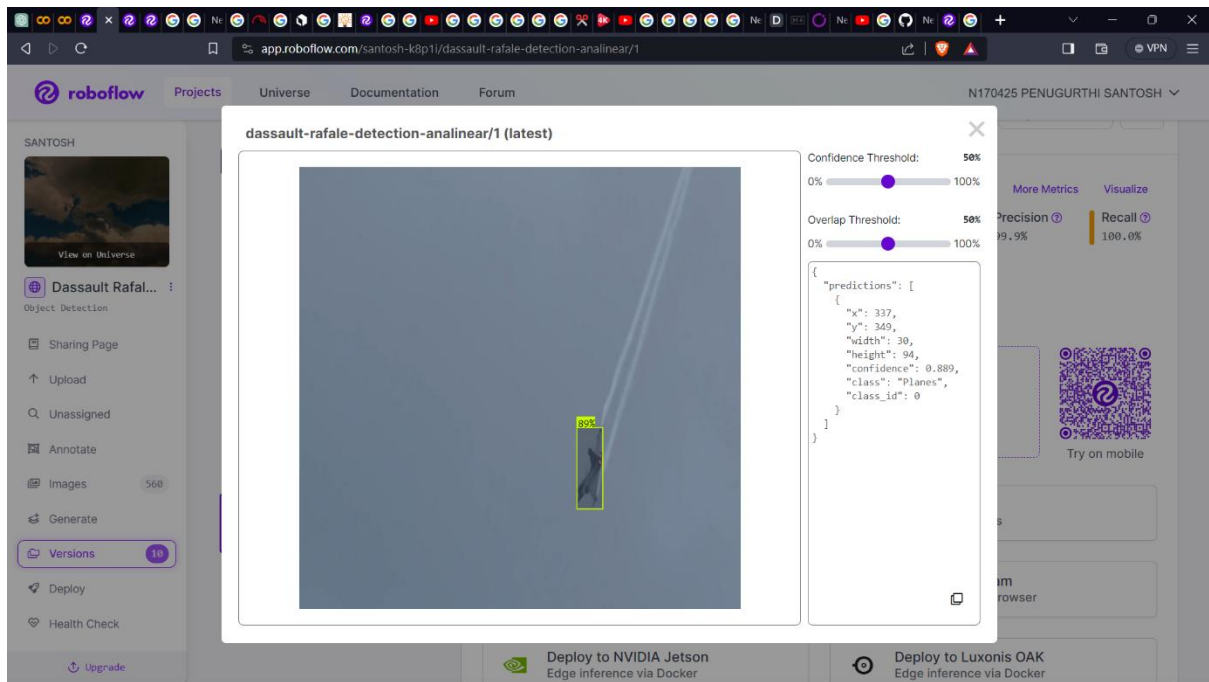


Figure 30. Testing Deployed Model in roboflow

5: CONCLUSION & FUTURE ENHANCEMENTS

5.1 Conclusion

In conclusion, this task aimed to demonstrate practical knowledge in computer vision through the implementation of object detection using YOLOv8. The selection of Dassault Rafale as the target object, coupled with the utilization of Google Colab for an accessible working environment and Roboflow for streamlined data collection and preparation, contributed to the successful realization of the task's objectives.

The YOLOv8 model proved to be a suitable choice, providing a balance between speed and accuracy. The implementation showcased the efficiency of the model in detecting and localizing Dassault Rafale instances in diverse scenarios. The use of a pre-trained model, combined with fine-tuning on the specific dataset, facilitated rapid development and deployment.

5.2 Advantages and Limitations

5.2.1 Advantages

- **Real-Time Object Detection:** YOLOv8 is designed for real-time object detection, providing fast and efficient inference speeds compared to some other object detection algorithms.
- **Single Forward Pass:** YOLO performs object detection in a single forward pass through the neural network, making it computationally efficient and suitable for real-time applications.
- **Unified Detection:** YOLO detects multiple objects in an image simultaneously, providing bounding box coordinates and class predictions for each detected object.

- **Architecture Flexibility:** YOLOv8 is flexible and allows users to choose different architectures and configurations based on their specific requirements. Users can select from different YOLO variants like YOLOv3 or YOLOv4.
- **Object Recognition Across Scales:** YOLO is designed to recognize objects of varying sizes and scales within an image, making it suitable for detecting objects at different distances from the camera.
- **Minimal False Positives:** YOLO algorithms are known for producing fewer false positives compared to some other object detection methods.
- **End-to-End Solution:** YOLO provides an end-to-end solution for object detection, including training and deployment, making it convenient for users who want a unified workflow.
- **Open-Source Implementation:** YOLO is implemented in an open-source framework, allowing researchers and developers to access the code, modify it, and integrate it into their projects.
- **Community Support:** YOLO has a large and active community, which contributes to ongoing improvements, updates, and the development of various applications.
- **Handling Multiple Objects:** YOLO is effective at handling scenarios where there are multiple objects of different classes in a single image.

5.2.2 Limitations

- **Fine Object Details:** YOLOv8, being a single-shot detector, may face challenges in capturing fine details of small objects compared to slower, region-based detectors. The YOLOv8 model may struggle to detect objects in cluttered scenes or when objects are partially occluded. Additionally, the model may have difficulty detecting small objects or objects with low contrast.
- **Handling Occlusions:** YOLO algorithms can struggle with object detection in scenarios where objects are heavily occluded or overlapping.
- **Training Data Quality:** The performance of YOLOv8 heavily relies on the quality and diversity of the training data. Insufficient or biased training data may result in suboptimal performance.
- **Resource Intensive:** Training YOLOv8, especially on large datasets or with high-resolution images, can be computationally intensive and may require powerful hardware.
- **Limited Context Understanding:** YOLO processes the entire image at once, which might limit its ability to understand contextual relationships between objects in a scene compared to region-based methods.
- **Handling Varied Aspect Ratios:** YOLOv8 may face challenges in accurately handling objects with extreme aspect ratios, such as very tall or wide objects.

5.3 Applications

The successful implementation opens avenues for various applications, including security surveillance, aviation maintenance, and educational purposes, military, where accurate detection of Dassault Rafale instances is essential. Let us see in detailed-

- **Military Surveillance:**
 - Utilize object detection on Dassault Rafale for military surveillance purposes. This can aid in monitoring and securing sensitive areas.
- **Aircraft Maintenance:**
 - Implement object detection for aircraft maintenance. Detecting specific parts or anomalies on Dassault Rafale can assist in maintenance tasks.
- **Security and Border Control:**
 - Use the trained model for border control and security applications, identifying aircraft at checkpoints or borders.
- **Airshow Event Management:**
 - Implement the model for managing airshow events, ensuring the accurate tracking and safety of Dassault Rafale during aerial displays.
- **Customs and Cargo Inspection:**
 - Apply object detection for cargo inspection at airports. Identify Dassault Rafale parts or equipment during customs checks.
- **Air Traffic Control Systems:**
 - Integrate the model into air traffic control systems to enhance the identification and monitoring of Dassault Rafale in controlled airspace.

5.4 Challenges Faced and Solutions

During the project, challenges were encountered, such as fine-tuning for specific object details and balancing accuracy with real-time processing. These challenges were addressed through iterative development, experimentation, and fine-tuning hyperparameters.

5.4.1 Data Collection Challenges:

Challenge: Limited Dassault Rafale Image Data: Acquiring a diverse dataset of Dassault Rafale images might be challenging due to limited availability.

Solution: Data Augmentation and Synthesis: Implement extensive data augmentation techniques to artificially increase the dataset, including rotation, flipping, scaling, and introducing variations in lighting and backgrounds.

5.4.2 Model Training Challenges:

Challenge: Model Convergence Issues: YOLOv8 model might face convergence challenges during training, leading to suboptimal results.

Solution: Hyperparameter Tuning and Transfer Learning: Experiment with different hyperparameters such as learning rates and batch sizes. Leverage transfer learning by initializing the model with pre-trained weights on a relevant dataset.

5.4.3 Model Validation Challenges:

Challenge: Overfitting or Underfitting: The model may exhibit overfitting or underfitting issues during validation, impacting its generalization to unseen data.

Solution: Regularization Techniques: Apply regularization techniques such as dropout or weight decay to mitigate overfitting. Adjust model complexity and dataset balance to address underfitting.

5.4.4 Deployment Challenges:

Challenge: Real-time Inference Performance: Achieving real-time object detection on new Dassault Rafale images may pose challenges in terms of model speed.

Solution: Model Optimization and Deployment Strategies: Optimize the YOLOv8 model for inference speed. Consider quantization, model pruning, or deploying on hardware accelerators for faster real-time performance.

5.5 Future Enhancements

The project lays the foundation for potential improvements and expansions:

- **Dataset Diversity:** Expanding the dataset to include a wider variety of Dassault Rafale instances and scenarios for enhanced model generalization.
- **Model Optimization:** Exploring model optimization techniques to further improve real-time processing speed.
- **Integration with Other Technologies:** Investigating integration with other technologies, such as geospatial data, to enhance contextual awareness.

6: REFERENCES

6.1 References

The success of this task was made possible through the integration of various tools and technologies. Key references include:

- **YOLOv8 Repository** : <https://github.com/ultralytics/ultralytics>
- **Roboflow** : <https://docs.roboflow.com>
- **Google Colab** : <https://colab.research.google.com>

Above references provided the necessary frameworks, platforms, and resources to implement and achieve the objectives outlined in this task.

Task Repositories and Workspaces:

- **Github Repository Link :**
<https://github.com/SantoshPenugurthi/Dassault-Rafale-Detection-Analinear>
- **Roboflow Project Link :**
<https://universe.roboflow.com/santosh-k8p1i/dassault-rafale-detection-analinear>
- **Google Colab Notebook:**
https://colab.research.google.com/drive/1fWfx9GSQ_5bQFKZxYRZ1eqUbE9XK_aun?usp=sharing