RF-DETR for Object Detection

This project guides you through building an object detection system using the RF-DETR (Receptive Field Enhanced Detection Transformer) model. RF-DETR is an advanced object detection model that allows a better detection performance to small or complex objects while maintaining the benefits of end-to-end optimization and simplicity [2].This activity will begin by acquiring and preparing a dataset, typically in COCO format, followed by preprocessing steps such as data augmentation and splitting into training, validation, and test sets. Next, you will implement the RF-DETR model in Python using a deep learning framework like PyTorch, integrating key components like a CNN backbone, transformer encoder-decoder, and the Receptive Field Enhancement module. After training the model, you will evaluate its performance using metrics such as accuracy, precision, recall, and mean average precision (mAP), followed by visualizing its predictions on test images.

By completing this project, you will gain hands-on experience in object detection, model implementation, and evaluation. You will also learn how to optimize the model through techniques like hyperparameter tuning and early stopping, while analyzing its strengths and weaknesses. Ultimately, this project will provide you with the skills to develop, train, and assess advanced object detection models for real-world applications. You may access the Google Colab Notebook, or follow the reference guide video.

I. Data Acquisition and Preparation

a. Dataset and Preprocessing

The Aquarium dataset from Roboflow contains 1,255 high-quality images of marine life distributed across 8 distinct classes including fish, jellyfish, penguins, puffins, sharks, starfish, and stingrays, with 1,171 training samples and 84 test samples. This dataset suits mechanism research for attention because it features diverse aquatic species with varying sizes, colors, textures, and shapes. The visual complexity and diversity of marine environments provide an excellent testbed for comparing how different attention mechanisms handle various visual features like fine details, large-scale patterns, and complex backgrounds [3]. The dataset's real-world nature, with natural lighting conditions and authentic aquarium settings, makes it more challenging and representative than synthetic datasets, allowing us to evaluate how attention mechanisms perform on practical classification tasks [4].

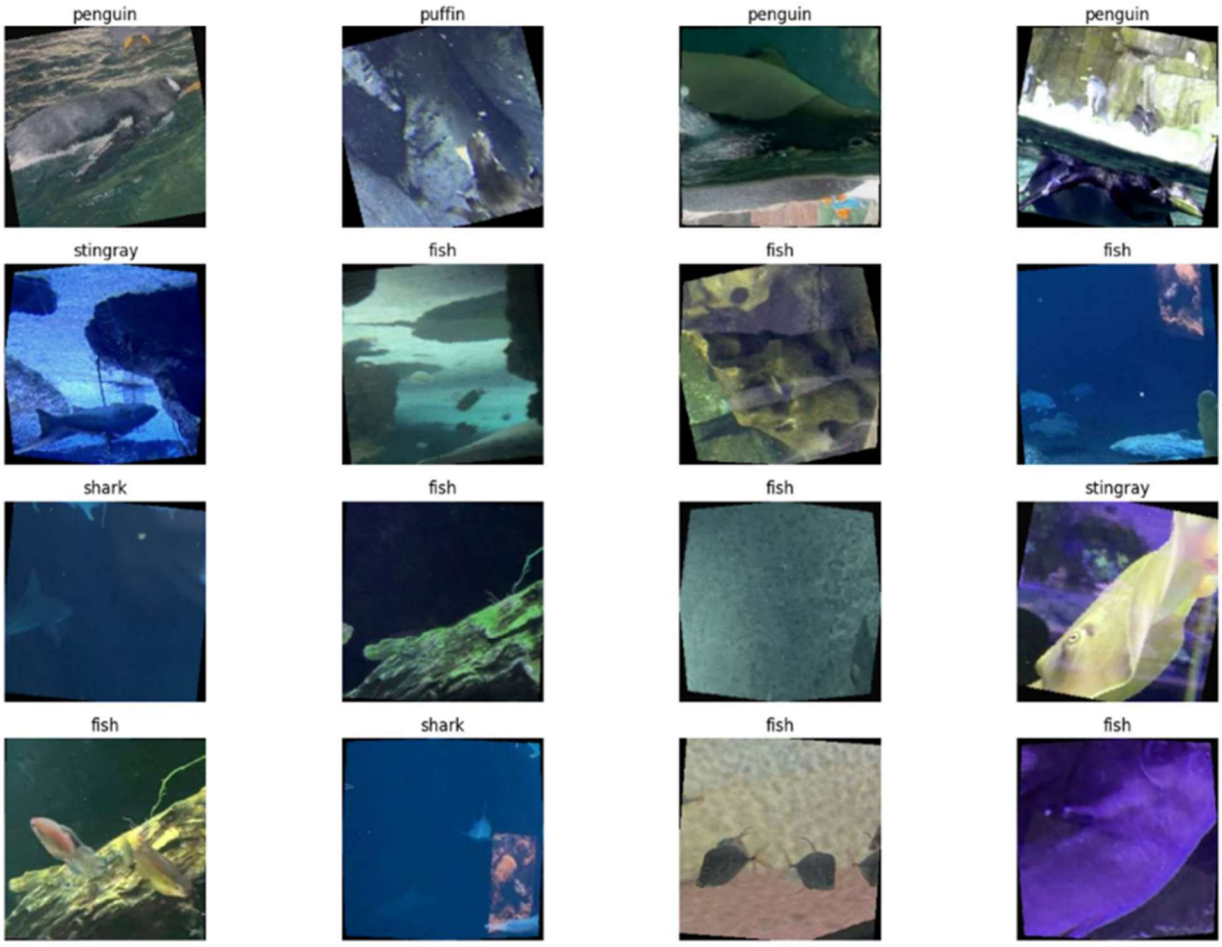


Figure 1. Visualization of the Data

b. *Environment Setup*

To begin with the environment setup, we will import the os and userdata modules. The line of code sets up an environment variable HF_TOKEN, which is retrieved from userdata. This token likely serves as an API key or access token for connecting to external services, such as Hugging Face for machine learning models or datasets [5]. This step ensures that the necessary credentials are available for API access in later stages of the project. By properly setting environment variables, the code avoids exposing sensitive data directly in the script, which is a good security practice. Then, we will check the status of the NVIDIA Tesla T4 GPU on Google Colab using the nvidia-smi command. It provides details about the GPU's current status, including its driver version, memory usage, and whether it is being actively utilized. In this case, the Tesla T4 is not running any processes, as indicated by the "No running processes found" message. This command is useful for monitoring the resources available for deep learning tasks, ensuring that the GPU is ready to be utilized for model training or inference. Additionally, it helps confirm that the CUDA environment is properly set up for computations.

```
[1] import os
    from google.colab import userdata

    os.environ["HF_TOKEN"] = userdata.get("HF_TOKEN")
```

```
[2] !nvidia-smi
```

```
Mon Jul 28 11:20:00 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       Off |   00000000:00:04.0 Off |                    0 |
| N/A   55C    P8             10W /   70W |       0MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

Following this is installing the required Python package rfdet and roboflow library which is used to interact with the Roboflow platform for dataset management. The api_key provided is required to authenticate access to a specific Roboflow workspace. After the library is installed, the code establishes a connection with the workspace, followed by retrieving the dataset version to ensure compatibility. This integration simplifies dataset management by allowing easy access to datasets and versions stored on Roboflow's cloud platform, facilitating the smooth setup of the project. Installing the correct dependencies is crucial for ensuring that the project runs smoothly, as missing libraries could lead to errors or unexpected behavior during training or evaluation.

```
[3] !pip install -q rfdetr
```

```
[4] !pip install roboflow

    from roboflow import Roboflow
    rf = Roboflow(api_key="AEQItibgYp6CszB4A1wr")
    project = rf.workspace("purin-rcwo6").project("aquarium-ea5ve-xraxm")
    version = project.version(9)
    dataset = version.download("coco")
```

```
[5] dataset.location
```

```
'/content/Aquarium-9'
```

Finally, the dataset.location line of code retrieves the location of the dataset, ensuring that the path to the dataset is correctly set up for use in later stages of the project. This setup step ensures that the

dataset is ready for downloading and use in training machine learning models. Proper dataset management is essential in machine learning projects, as it affects data loading, preprocessing, and model training efficiency.

II. RF-DETR Model Implementation

a. Initializing RF-DETR Model and Callback Function

This code snippet initializes an RF-DETR model and defines a callback function to store training data. The RFDETRBase class is imported from the rfdetr package and is instantiated to create the model object. A history list is created to store data related to training, like loss and other metrics. The callback function callback2 appends the incoming data to the history list after each epoch ends. This callback is then added to the model's on_fit_epoch_end callback list. This is a simple method to track and store important metrics for later analysis, especially when training deep learning models like RF-DETR.

```
from rfdetr import RFDETRBase

model = RFDETRBase()
history = []

def callback2(data):
  history.append(data)

model.callbacks["on_fit_epoch_end"].append(callback2)

model.train(dataset_dir=dataset.location, epochs=30, batch_size=4, grad_accum_steps=4, lr=1e-4)
```

b. Predicting and Annotating Images

Here, the RF-DETR model is being used for predictions on an image from the dataset. The model's output is processed by the supervision library to annotate and visualize the results. The code calculates the optimal scale for text and line thickness when displaying the bounding boxes and labels. Then, it proceeds to annotate the image with bounding boxes and labels showing the detected objects. These annotated images are displayed side-by-side with the original images to visually compare the ground truth (annotations) and the model's detections, which is crucial for model evaluation.

```
[ ]  from rfdetr import RFDETRBase
     import supervision as sv
     from PIL import Image

     path, image, annotations = ds[0]
     image = Image.open(path)

     detections = model.predict(image, threshold=0.5)

     text_scale = sv.calculate_optimal_text_scale(resolution_wh=image.size)
     thickness = sv.calculate_optimal_line_thickness(resolution_wh=image.size)

     bbox_annotator = sv.BoxAnnotator(thickness=thickness)
     label_annotator = sv.LabelAnnotator(
         text_color=sv.Color.black(),
         text_scale=text_scale,
         text_thickness=thickness,
         smart_positioning=True
     )

     annotatios_labels = [
         f"{ds.classes[class_id]}"
         for class_id
         in annotations.class_id
     ]

     detections_labels = [
         f"{ds.classes[class_id]}"
         for class_id, confidence
         in zip{detections.class_id, detections.confidence}
     ]

     annotation_image = image.copy()
     annotation_image = bbox_annotator.annotate(
         annotation_image, annotations
     )
     annotation_image = label_annotator.annotate(
         annotation_image, annotations, annotation_labels
     )

     detection_image = image.copy()
     detection_image = bbox_annotator.annotate(
         detection_image, detections
     )
     detection_image = label_annotator.annotate(
         detection_image, detections, detections_labels
     )

     sv.plot_images_grid(images=[annotation_image, detection_image], grid_size=(1, 2), titles=["Annotation", "Detection"])
```

a. Preparing Dataset for Evaluation

This snippet prepares the dataset for evaluation by collecting predictions and comparing them against the targets. It iterates through the dataset, makes predictions for each image, and stores both the predicted detections and the ground truth annotations in lists. These lists are then used for performance metrics such as mean average precision (mAP). This process allows the model's predictions to be evaluated against the actual object annotations, providing an objective measure of its accuracy and effectiveness.

```
[ ]  import supervision as sv
     from tqdm import tqdm
     from supervision.metrics import MeanAveragePrecision

     targets = []
     predictions = []

     for path, image, annotations in tqdm(ds):
       image = Image.open(path)
       detections = model.predict(image, threshold=0.5)

       targets.append(annotations)
       predictions.append(detections)
```

III. Model Training and Optimization

a. Plotting Training and Validation Loss

This part of the code is used for plotting the training and validation loss over the epochs. The history list, which contains the tracked metrics, is converted into a pandas DataFrame for easier manipulation. Using Matplotlib, the code then generates a plot showing how both the training loss and validation loss change with each epoch. The train_loss and val_loss values are plotted against the epoch values, helping to visualize the learning curve of the model during training. This kind of plot is essential to monitor overfitting or underfitting in the model. The labels and grid are added for better clarity and understanding.

```
[ ]  import matplotlib.pyplot as plt
     import pandas as pd

     df = pd.DataFrame(history)
     plt.figure(figsize=(12, 8))

     plt.plot(
         df['epoch'],
         df['train_loss'],
         label='train_loss'
         marker='o',
         linestyle='-'
     )

     plt.plot(
         df['epoch'],
         df['val_loss'],
         label='val_loss'
         marker='o',
         linestyle='--'
     )


     plt.xlabel('Epoch')
     plt.ylabel('Loss')
     plt.title('Training and Validation Loss')
     plt.legend()
     plt.grid(True)

     plt.show()
```

a. Plotting Average Precision and Recall

In this code, the focus is on plotting additional performance metrics, specifically average precision and average recall. The history data frame is again used, but this time new columns for average precision (avg_precision) and average recall (avg_recall) are calculated from the test evaluation results. The code then plots these metrics over each epoch using Matplotlib. The plots are separately labeled with appropriate titles and axes labels to clarify what the viewer is seeing. These metrics are important for evaluating object detection models, as they show the balance between precision and recall during model performance.

```
[ ]  import matplotlib.pyplot as plt
     import pandas as pd

     df = pd.DataFrame(history)

     df['avg_precision'] = df['test_eval'].apply(lambda arr: arr[0])
     df['avg_recall'] = df['test_eval'].apply(lambda arr: arr[6])

     plt.figure(figsize=(12, 8))
     plt.plot(
         df['epoch'],
         df['avg_precision'],
         label='avg_precision'
         marker='o',
         linestyle='-'
     )

     plt.title('Average Precision')
     plt.xlabel('Epoch')
     plt.ylabel('Average Precision')
     plt.legend()
     plt.grid(True)

     plt.show()

     plt.figure(figsize=(12, 8))
     plt.plot(
         df['epoch'],
         df['avg_recall'],
         label='avg_recall'
         marker='o',
         linestyle='-'
     )

     plt.title('Average Recall')
     plt.xlabel('Epoch')
     plt.ylabel('Average Recall')
     plt.legend()
     plt.grid(True)

     plt.show()
```

IV. Model Evaluation

a. Loading Dataset in COCO Format

This image shows code related to working with a dataset in the COCO format. The supervision library is used to load the detection dataset from COCO annotations into a structured dataset object (ds). The dataset contains both image files and their associated annotations. The path to the test images and annotations are passed to the DetectionDataset.from_coco() method, which is a useful utility for organizing and accessing the data required to train or evaluate object detection models. This is an important step as it allows the model to learn from properly formatted data, improving its accuracy.

```
[ ]  import supervision as sv

     ds = sv.DetectionDataset.from_coco(
         images_directory_path=f"{dataset.location}/test",
         annotations_path=f"{dataset.location}/test/_annotations.coco.json"
     )
```

b. Computing and Plotting Mean Average Precision

In this code, the mean average precision (mAP) metric is updated and computed. The MeanAveragePrecision() function is used to assess how well the model performs in detecting objects across all classes in the test set. The computed result is then plotted using the plot() function. This is an essential step in evaluating the model's overall performance, as mAP is a widely used metric in object detection tasks. The plot gives a clear view of how well the model detects objects, and it can be used to compare different models or configurations.

```
[ ]  map_metric = MeanAveragePrecision()
     map_result = map_metric.update(predictions, targets).compute()

     map_result.plot()
```

c. Plotting Confusion Matrix

This final image shows how to calculate and plot the confusion matrix for the object detection model. The ConfusionMatrix.from_detections() function from the supervision library takes the list of predictions and the ground truth targets to create a confusion matrix. The matrix is then plotted, providing a visualization of how many predictions were correct, how many were false positives, and how many were false negatives. This helps in understanding the performance of the model on a class-by-class basis and is critical for diagnosing where the model needs improvement.

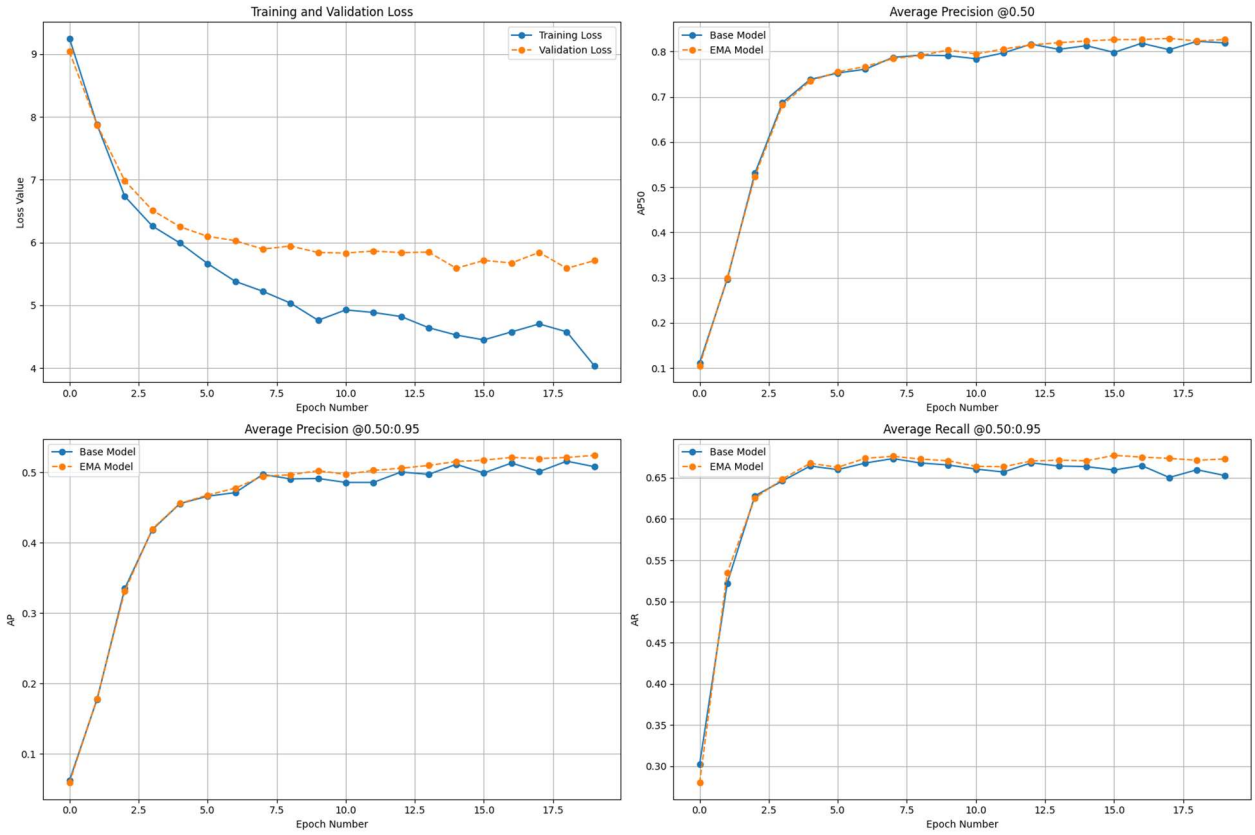
```
[ ]  confusion_matrix = sv.ConfusionMatrix.from_detections(
         predictions=predictions,
         targets=targets,
         classes=ds.classes
     )

     _ = confusion_matrix.plot()
```

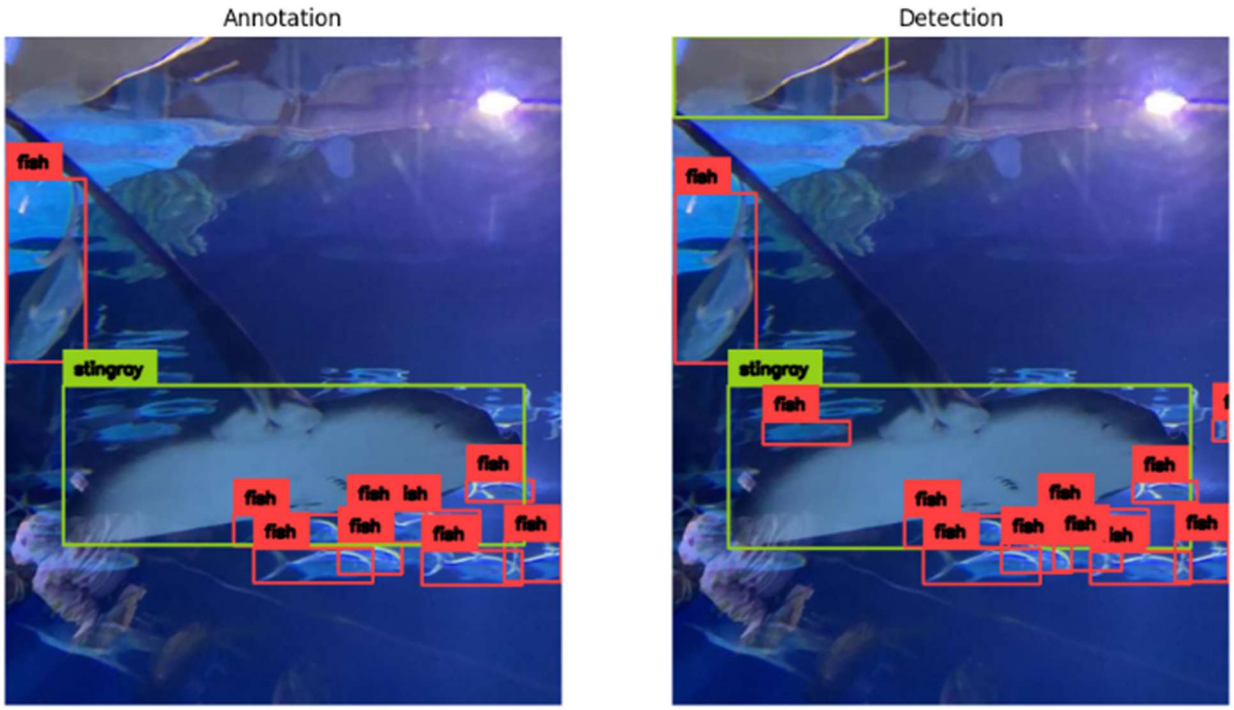
V. **Results**

The first set of plots shows how the model's performance evolves over the epochs. The "Training and Validation Loss" graph shows that the training loss decreases, indicating that the model is learning, while the validation loss fluctuates but generally decreases as well. In the "Average Precision" and "Average Recall" plots, the curves for both the base model and the enhanced model

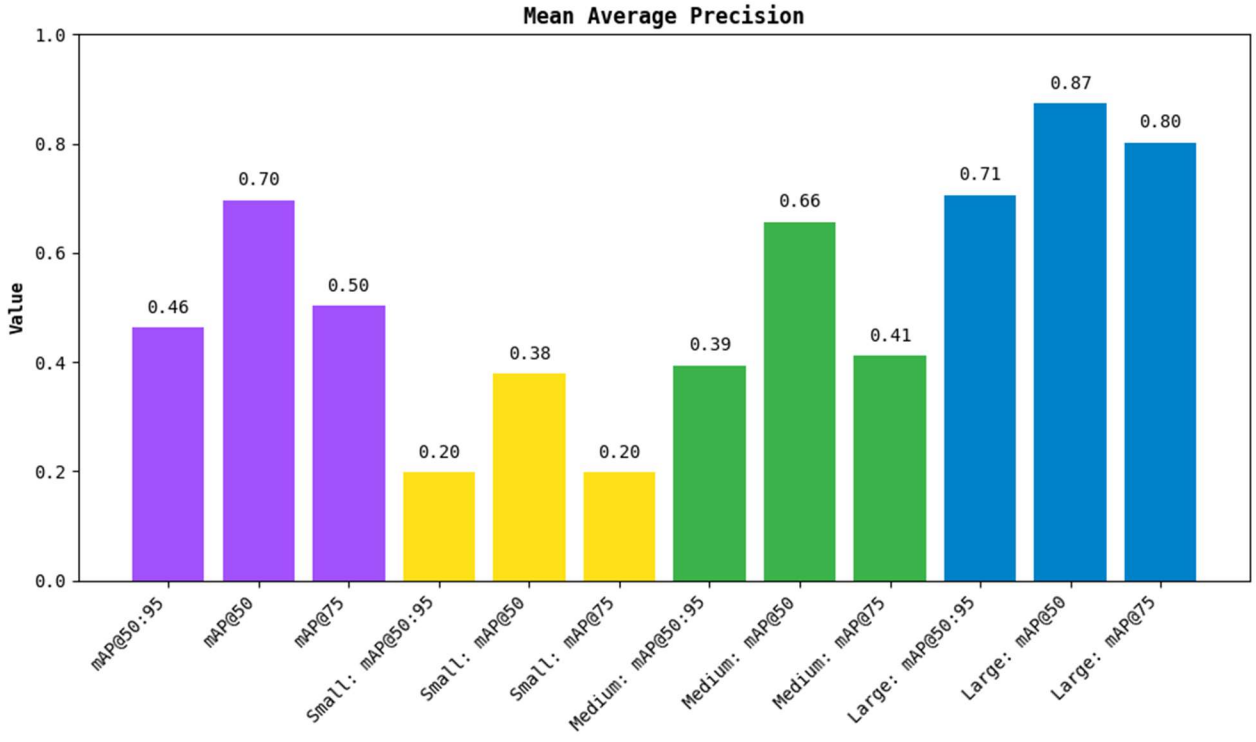
show that precision and recall steadily improve, indicating better detection accuracy as training progresses.



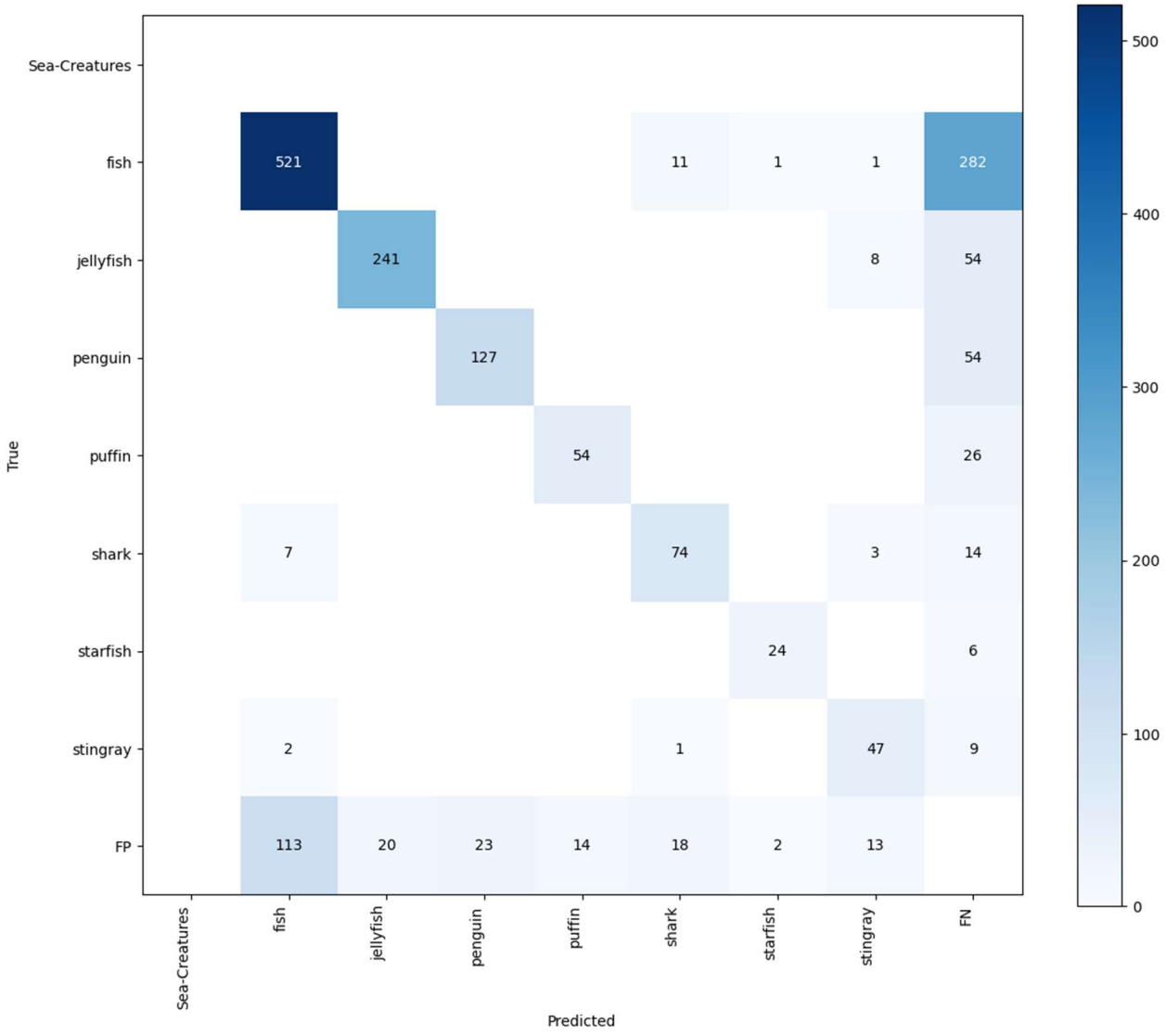
This image shows a comparison between ground truth annotations (left) and model predictions (right) for a sea creature image. The red bounding boxes on the left represent the true labels of different sea creatures, while the green and red bounding boxes on the right represent the predicted labels. The model detects the "fish" and "stingray" with varying levels of accuracy.



This plot shows the mean average precision (mAP) scores for various object sizes (small, medium, large). The mAP values for different size classes are compared, showing that the model performs better on larger objects (e.g., "Large: mAP@0.75"), while its performance on smaller objects (e.g., "Small: mAP@0.5") is less accurate.



The confusion matrix shows the performance of the model in predicting different sea creatures. The matrix shows the true class labels (on the left) against the predicted class labels (on the top), with the darker colors representing higher counts. The model has difficulty distinguishing between certain classes like "fish" and "stingray," as indicated by the higher number of false positives (FP) in those categories.



This table displays the validation results for each class, showing the mAP scores at both 0.5 and 0.95 IoU thresholds, as well as the precision and recall values. The results show consistent performance across the classes with an overall mAP of around 0.74 for most classes, suggesting that the model is performing well on the validation dataset.

Validation Results

	class	map@50:95	map@50	precision	recall
1	fish	0.4424734900935351	0.7539150795150764	0.7181818181818181	0.74
2	jellyfish	0.6440211813726502	0.9728804021504901	1.0	0.74
3	penguin	0.39232729280983547	0.7541182656202104	0.7024793388429752	0.74
4	puffin	0.3682674772350037	0.6672250937846091	0.5714285714285714	0.74
5	shark	0.5033097524982416	0.827910783849603	0.8490566037735849	0.74
6	starfish	0.6922940248044627	0.9381056821355982	1.0	0.74
7	stingray	0.6272912698116677	0.8714152284703618	0.8666666666666667	0.74
8	all	0.5242834983750567	0.826510076503707	0.8154018569848025	0.74

Similar to the validation results table, this one shows the test set performance, with mAP scores at the same IoU thresholds and precision and recall values for each class. The model performs slightly better on the test set compared to the validation set, with mAP values ranging from 0.66 to 0.76 across different classes.

Test Results

	class	map@50:95	map@50	precision	recall
1	fish	0.42776829469137684	0.7629108150289792	0.6550632911392406	0.76
2	jellyfish	0.5962597544643502	0.8937685870311288	0.9437751004016064	0.76
3	penguin	0.3818643463275697	0.7645004490276444	0.6731707317073171	0.76
4	puffin	0.406186534947812	0.7687325405732017	0.7654320987654321	0.76
5	shark	0.5150982110382594	0.7856746430946614	0.7575757575757576	0.76
6	starfish	0.6903953791989154	0.8843982537211539	0.9583333333333334	0.76
7	stingray	0.5993072032354063	0.7679566350822032	0.75	0.76
8	all	0.5166971034148128	0.8039917033655676	0.7861929018460981	0.76

VI. Discussion of Results

The object detection model produced results demonstrating very good performance across various classes of sea creatures. Looking at the aspect of loss, both training and validation losses decrease with epochs, which is a good sign that the model is learning well with data. The further positive trend in the average precision and recall metrics supports this and confirms that there has been a steady improvement in detection accuracy. Big objects are well detected by the model, reflecting in higher mAP values for larger species of sea creatures such as fish and stingrays. In contrast, smaller objects such as jellyfish are detected with lower precision and recall due to the common challenges faced while detecting small objects in object detection tasks. The confusion matrix confirms some of these challenges, especially with certain categories exhibiting a large number of false positives, which means that the model is likely to confuse similar objects among different types of fish.

In terms of evaluation, the validation results show a steady mAP score of around 0.74 for most classes, while the test results reveal a slight improvement, with mAP values reaching up to 0.76. This suggests that the model generalizes fairly well from the validation set to unseen test data. The model's ability to maintain consistent precision and recall values across different sea creature categories indicates that the network is robust, though there is still room for improvement, especially for smaller object classes. Overall, these results demonstrate a strong foundation for the model, with room to optimize its performance, particularly on smaller objects, possibly by fine-tuning the model or incorporating additional data augmentation techniques for better generalization.

VII. Future Work and References

[1] https://www.youtube.com/watch?v=dvOK-nYtrGI
[2] https://pypi.org/project/rfdetr/
[3] https://universe.roboflow.com/microsoft/coco
[4] https://medium.com/red-buffer/roboflow-d4e8c4b52515
[5] https://www.thecloudgirl.dev/blog/complete-beginners-guide-to-hugging-face-2025