Sea SAR Detectors

Object Detection for Maritime Search and Rescue using Deep Learning

X

21/9/2025

Contents

Introduction

The aim of this project is to explore and compare different deep learning approaches for maritime object detection using drone imagery. We work with the SeaDronesSee Object Detection v2 (Compressed)which contains challenging sea scenarios where small objects must be detected under varying conditions. Our study covers a range of representative object detectors that reflect the progression of the field: **YOLOv1** as an early single-stage baseline, **Faster R-CNN** as a two-stage detector, **RetinaNet** as a one-stage method addressing any class imbalance, **YOLOv8** as a modern real-time architecture, and **DETR** as a transformer-based approach. By benchmarking these models on the same dataset, we aim to highlight how detection methods have advanced over time and to assess their suitability for real-world search and rescue applications.

How to Run the Project

The project is on GitHub: https://github.com/GiX007/sea-sar-detectors. It was developed on Windows and also runs on macOS.

Requirements and Installation

The project requires **Python 3.10+**. All required packages are listed in **requirements.txt** (including NumPy, Pandas, Matplotlib, PyTorch, Torchvision, and others). To setup the requirements, run:

git clone https://github.com/GiX007/sea-sar-detectors.git

cd sea-sar-detectors

pip install -r requirements.txt

Hardware: All experiments were run with **PyTorch 2.5.1+cu121** on a single **NVIDIA GeForce GTX 1050 Ti** (4 GB VRAM, CUDA 12.1, cuDNN 9.1).

Project Structure

The structure of the project:

- Sea SAR Detectors Project
    - data/
    - src/
        - models/
            - yolo_v1.py
            - faster_rcnn.py

        - retinanet.py
        - yolo_v8.py
        - detr.py
      - eda.py
      - preproces.py
      - utils.py
    - main.py
    - results/
    - README.md
    - requirements.txt
    - predict_commands.txt

src/ contains the core code (EDA, preprocessing, all models, and utilities) and the main script is **main.py**.

results/ stores metrics, plots and predictions.

data/ is where the dataset is.

The project also includes **README.md**, and a **predict_commands.txt**.

Running

To run the full pipeline (EDA → preprocessing → training → evaluation) execute:

python main.py

To run inference on a single image with a trained model (example with Faster R-CNN) execute:

python –m src.predict_single `

–-model_type fasterrcnn `

--model_path results/models/fastserrcnn.pth `

--image_path data/test_images/209.jpg `

--class map results/models/class_map.json `

--score_thr 0.6

Additional ready-to-use commands for single-image inference with different models, images, and thresholds are provided in ***predict_commands.txt****.*

Dataset

The project uses the **SeaDronesSee Object Detection v2 dataset**, published by the **University of Tubingen (Germany)** as a benchmark for maritime search and rescue. The dataset is organized into three splits with **8,930 training images**, **1,547 validation images**, and **3,750 test images**, stored in high-resolution **.jpg** format. Images come from drones over open sea and coastal scenarios, with diverse resolutions (most commonly 2160×3840 and 1080×1920) and consistent 3-channel RGB encoding.

Annotations follow the **COCO format** with JSON files defining categories, bounding boxes, and metadata. Five categories (classes) are provided: *swimmer, boat, jetski, life_saving_appliances, and buoy*. In total, the training set contains **57,760 annotated bounding boxes**, with objects often appearing as small targets against sea backgrounds. Each image record includes drone metadata such as GPS location, datetime, altitude, and camera orientation, providing additional context for analysis.

This dataset poses significant challenges for detection due to **small object sizes, high variability in backgrounds, and diverse recording conditions**, making it well-suited for evaluating modern object detection models.

EDA and Re-splitting

EDA

Before training, we performed an exploratory data analysis (EDA) of the dataset. The images are already cleaned and stored in standard **.jpg** format, but they come in **different shapes and resolutions** (for example, 2160x3840 and 1080x1920 are the most common). This variability is typical of drone footage. All images are **RGB** with consistent **uint8** pixel values, and the annotations are provided in **COCO JSON format** with bounding boxes and categories. A few sample images were visualized to check object sizes, and as expected, most targets appear very small compared to the full frame, making the detection problem challenging. To illustrate this, we first show original full-frame drone images and then zoomed-in views with annotations, highlighting how tiny and hard-to-spot the objects can be.

All details from the EDA process, re-splitting procedure, and preprocessing steps are summarized in **results/data_summary.txt**.

Figure 1: Original full-frame drone images from the SeaDronesSee dataset.

Figure 2: Zoomed-in annotated images showing swimmers and other objects.

Re-splitting and preprocessing

The dataset providers only annotations for the training and validation splits, while the official test set has no ground truth. To enable a fair evaluation, we **combined the original training and validation sets** into a single annotated pool and then created new **train/validation/test splits**. Some of the original test images without annotations are still used later for **single-image prediction examples**.

After re-splitting the annotated pool, the **new dataset sizes** were: **6,576 training images**, **1,139 validation images**, and **2,762 test images**. However, training all detectors on the full split would be very time-consuming given the large model architectures. To make the experiments feasible, we created **reduced DataLoaders** with around **658 training**, **114 validation**, and **690 test samples**, which allowed us to speed up the training and evaluation process while still maintaining representative class distributions.

For training deep learning models, it was also necessary to **resize all images to a fixed resolution** so that they could be efficiently batched. To speed up training and reduce memory use, we downscaled the images from their original high resolutions to **192×192 pixels**, which still preserves enough detail for benchmarking different detectors.

This re-splitting and resizing process ensures that all models are trained on the same data under consistent conditions, while still allowing us to test them on realistic unseen images.

Models

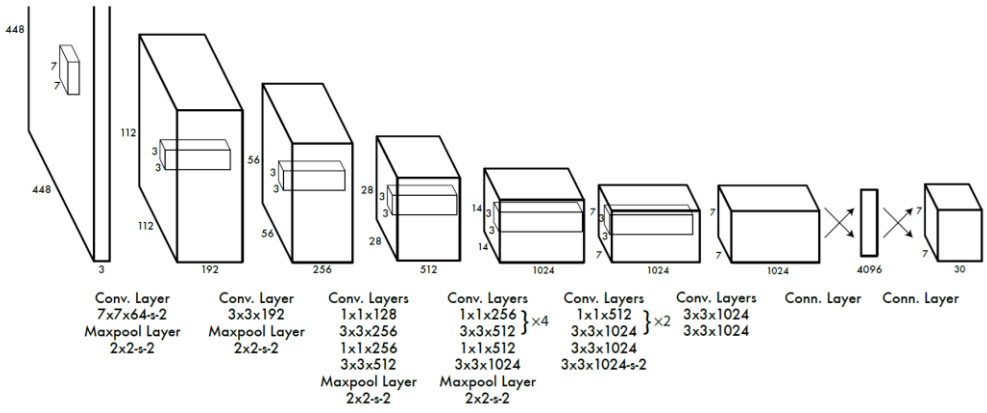
YOLOv1

Description

YOLOv1 *(J.Redmon, 2016 [1])* was one of the first models to frame object detection as a **single regression problem**. Instead of generating region proposals, the image is divided into a grid, and each grid cell directly predicts bounding box coordinates and class probabilities. This design made YOLOv1 extremely fast compared to earlier two-stage detectors, though its accuracy is limited for small objects.

Model Structure

Figure 3: YOLOv1 architecture (2016), showing the convolutional backbone and fully connected detection head.



Implementation

We implemented a compact version of **YOLOv1 from scratch in PyTorch**. The backbone follows the Darknet-style sequence of convolution and pooling layers, but with reduced depth: the repeated convolutional blocks were shortened (from 4 to 2). In addition, the original two large fully connected layers were replaced by a global average pooling followed by a lightweight linear layer. These modifications were introduced to reduce model size, speed up training, and make the architecture more practical for our dataset. This adaptation preserves the original YOLOv1 design, predicting bounding boxes and class scores from an S×S grid with B=2 boxes per cell, while making training feasible on our setup.

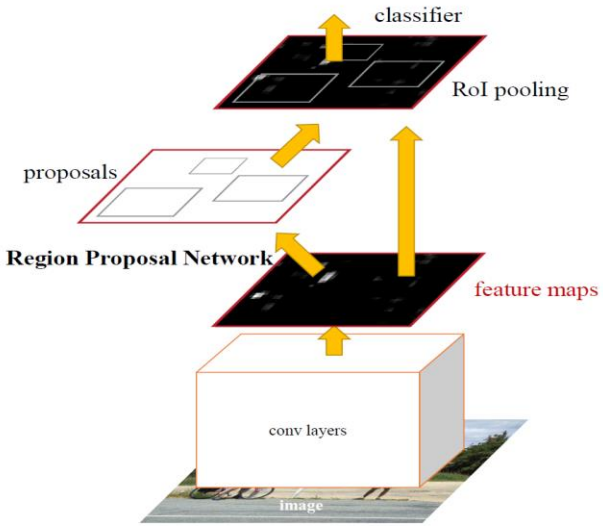
Faster R-CNN

Description

Faster R-CNN *(S.Ren, 2015 [2])* is a **two-stage object detector**. First, a **backbone CNN** extracts feature maps from the image. Then a **Region Proposal Network (RPN)** slides over these maps and suggests candidate bounding boxes where objects might be. Each proposed region is passed through **RoI pooling**, which crops the corresponding part of the feature map and resizes it to a fixed size. This ensures that the next stage can process all regions consistently, regardless of their original scale. Finally, a **classifier and box regressor** decide what object is inside each region and adjust the box coordinates. This design is slower than one-stage models like YOLO, but it is generally more accurate and reliable, especially for small or overlapping objects.

Model Structure

Figure 4: Faster R-CNN architecture (2015). The backbone CNN extracts feature maps, the Region Proposal Network (RPN) generates candidate regions, RoI pooling resizes each region to a fixed size, and the classifier predicts the object class and refined bounding box.



Implementation

We used the **torchvision implementation of Faster R-CNN** with a **MobileNetV3-Large + FPN backbone** pretrained on COCO. The default classification head is designed for COCO's 91 categories, which do not match our dataset. Therefore, we replaced it with FastRCNNPredictor(in_features, num_classes), so the model learns to predict our own classes instead.

Since the original Faster R-CNN is computationally heavy, we introduced a few modifications to make training feasible on our hardware:

- Reduced the number of region proposals passed to NMS (_pre_nms_top_n and _post_nms_top_n), (NMS = Non-Maximum Suppression, a step that removes overlapping boxes and keeps only the best candidates)
- Lowered the number of samples per image used for loss computation (batch_size_per_image in RPN and ROI heads)
- Kept the backbone frozen (trainable_backbone_layers=0) to save memory and stabilize training

These changes do not alter the core Faster R-CNN pipeline but significantly reduce computation time, making it practical for our experiments. Note that the use of an **FPN (Feature Pyramid Network)** was not part of the original Faster R-CNN paper but is included by default in torchvision's implementation to improve multi-scale detection.

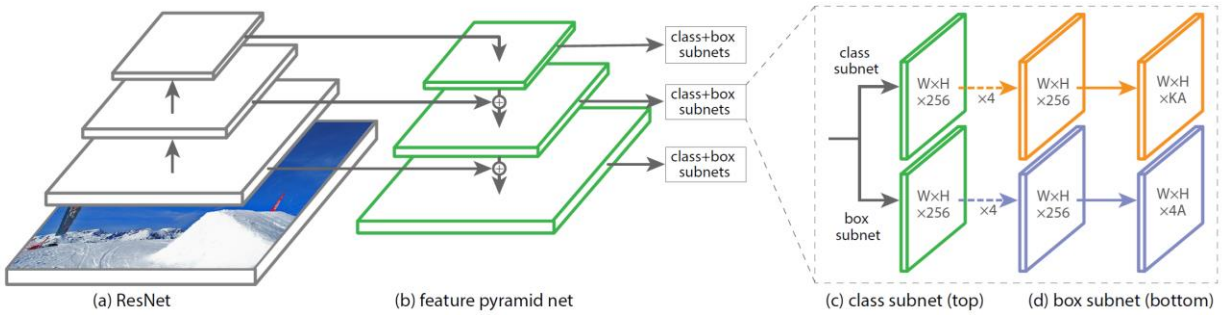
RetinaNet

Description

RetinaNet *(Lin, 2017 [3])* is a **one-stage object detector** designed to balance speed and accuracy. Unlike two-stage detectors such as Faster R-CNN, which uses a two-stage pipeline, RetinaNet directly predicts object classes and bounding boxes from a dense set of anchors spread across the image. Its key innovation is the **Focal Loss** (see more in [3]), which reduces the impact of easy background examples and focuses training on the harder, less frequent objects. This makes RetinaNet particularly effective for datasets with class imbalance.

Model Structure

Figure 5: RetinaNet architecture (2017). A ResNet backbone with a Feature Pyramid Network (FPN) feeds two subnetworks: one for classification (with focal loss) and one for bounding box regression.



Implementation

We used the **torchvision RetinaNet implementation** with a **ResNet-50 + FPN backbone** pretrained on ImageNet. The default COCO head was disabled (weights=None), and a fresh classification head was created with the correct number of classes for our dataset. To keep the model efficient, the ResNet backbone was frozen, and inference-time parameters were adjusted: the score threshold was increased (0.2 vs. default 0.05) to suppress low-confidence boxes and the maximum detections per image was reduced (100 vs. default 300) for faster evaluation. These adaptations helped stabilize training and speed up evaluation, making RetinaNet practical for our drone imagery dataset with many small, hard-to-detect objects.

Because RetinaNet uses label IDs starting at 0 (for foreground classes only), while our dataset uses 1-based labels with 0 reserved for background, we implemented an **adapter module** to automatically shift labels during training and evaluation. This ensured compatibility across models without changing the rest of the pipeline.

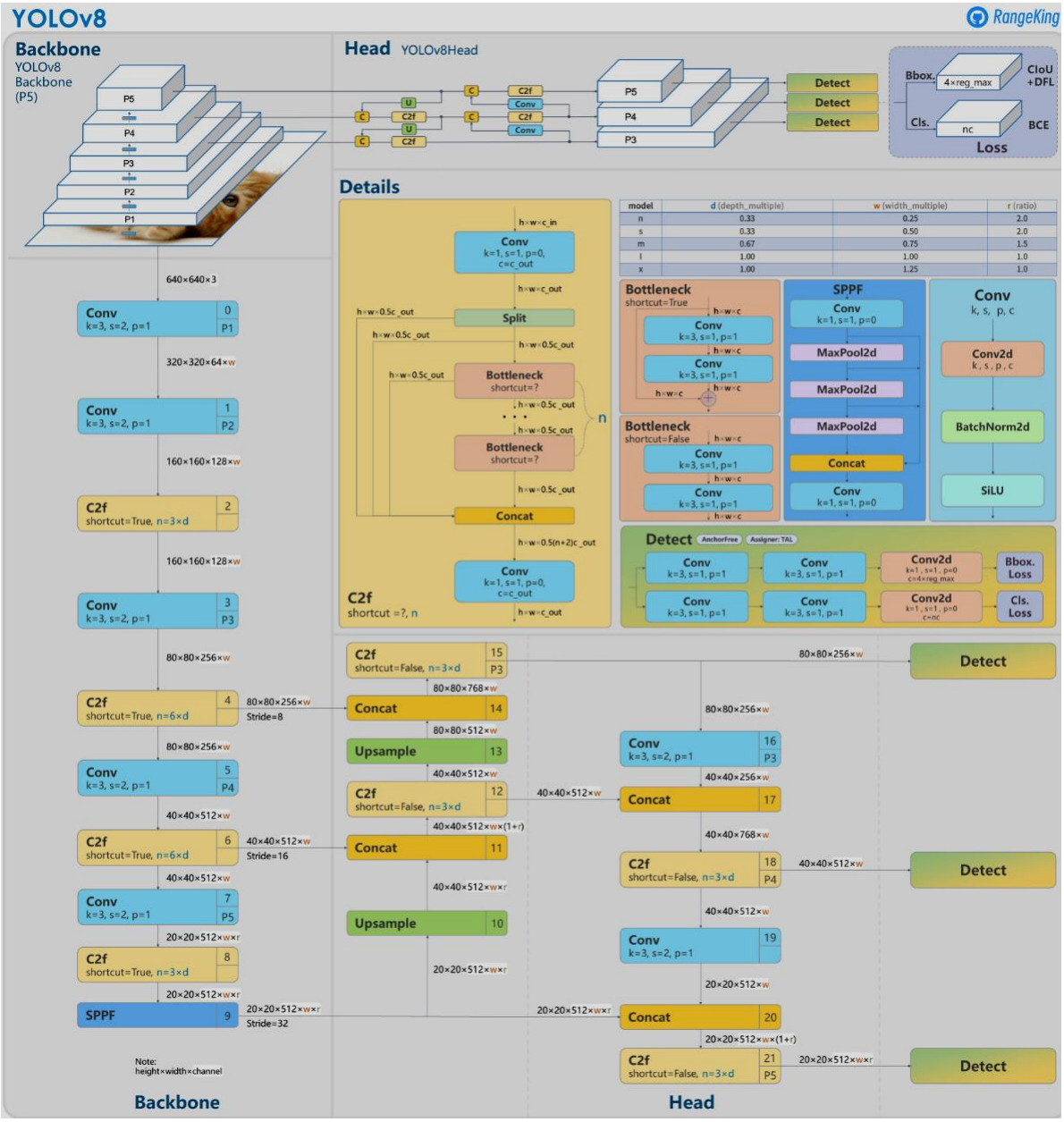
YOLOv8

Description

YOLOv8 *(Ultralytics, 2023 [4])* is the most recent version in the YOLO family of **one-stage detectors**. It builds on the speed and simplicity of earlier YOLO models while improving accuracy with a modern backbone and detection head. Unlike older YOLO versions that rely on predefined anchors, YOLOv8 uses an **anchor-free design**, predicting object centers and bounding box dimensions directly. It also comes in multiple sizes (from small to large), making it flexible for different hardware setups and applications.

Model Structure

Figure 6: YOLOv8 architecture (Ultralytics, 2023). The backbone extracts features, which are aggregated through a neck and passed to an anchor-free detection head for class and bounding box predictions.



Implementation

We did **not re-implement YOLOv8**, we used the official **Ultralytics** package (ultralytics) and trained a ready model with our data. Specifically, we a) converted COCO JSON annotations to YOLO **txt** labels and wrote a data.yaml describing our train/val/test folders and class names, b) trained

YOLOv8s (the small model) using the built-in train/val routines, and c) exported the best weights and logged training curves from results.csv. For fair comparison with our other models, we post-processed predictions into the same **boxes/labels/scores** format and evaluated on our test split, also reporting IoU/Precision/Recall@0.5. We slightly raised the **score threshold** (0.30) to reduce low-confidence boxes in cluttered scenes.

Note: the figure above shows many architectural details, **we did not modify those internals**, we relied on the official YOLOv8 model and defaults, only adjusting dataset formatting and a few inference/training settings.

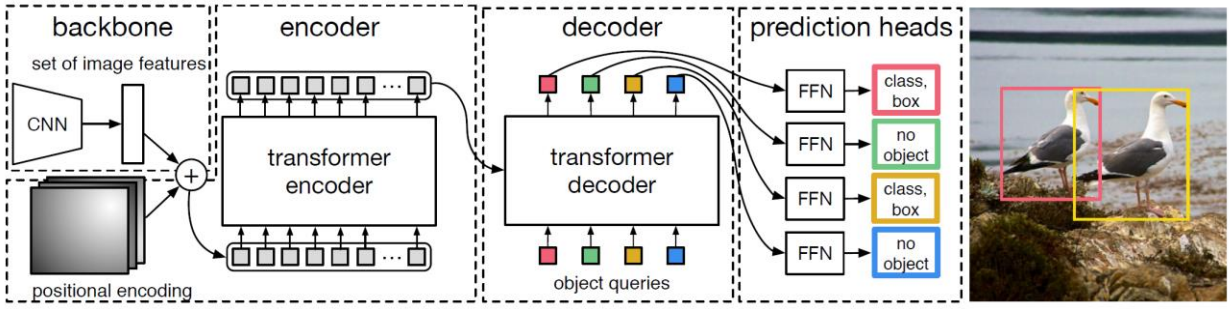
DETR

Description

DETR *(Carion, 2020 [5])* introduced a new approach to object detection by replacing hand-designed components (anchors, NMS, RPN) with a **Transformer**. An image is first processed by a CNN backbone to produce feature maps, which are flattened and passed to an encoder–decoder Transformer. The decoder works with a fixed set of *object queries*, each predicting either one object or "no object." DETR learns detection as a direct set prediction problem, using the Hungarian algorithm to match predictions to ground truth. This eliminates the need for anchors and post-processing, simplifying the detection pipeline.

Model Structure

Figure 7: DETR architecture (2020). Features from a CNN backbone are passed through a Transformer encoder–decoder. A fixed number of object queries are decoded into predictions, where each query outputs either an object's class and box or 'no object'.



Implementation

We used the **Hugging Face Transformers** library implementation of DETR (facebook/detr-resnet-50). The backbone CNN (ResNet-50) is pretrained on ImageNet, while the Transformer layers follow the original design with 6 encoder and 6 decoder blocks. For our dataset, we replaced the final prediction head to match the number of object categories. Training followed the standard DETR procedure with set-based loss combining classification and box regression. For evaluation,

we used the Hugging Face DetrForObjectDetection interface, ensuring predictions were converted to the same format as our other models for consistency.

Training Setup

All models were trained on the same dataset split (train/val/test) to ensure a fair comparison. For consistency, we followed a straightforward training procedure without hyperparameter tuning:

- **Optimizers and Learning Rates** – Models from the torchvision library (e.g., Faster R-CNN, RetinaNet) were trained with SGD (momentum = 0.9, weight decay = 0.0005, learning rate = 0.005). YOLOv1 used a similar setup, while YOLOv8 relied on the default Ultralytics training pipeline with AdamW.
- **Epochs and Early Stopping** – Each model was trained for up to 100 epochs with early stopping based on validation IoU@0.5 to prevent overfitting (for Detr, we set 200 epochs).
- **Batch Sizes and Image Sizes** – Due to hardware constraints, batch sizes were kept small (2 for all models and 4 for YOLOv8) and image sizes were fixed at **192×192** (after experimenting with larger resolutions such as 640×640, we found that reducing to 192×192 provided a reasonable balance between performance and training speed). For Faster R-CNN and RetinaNet, proposal counts were reduced to speed up training.
- **Implementation Choices** – We used simplified implementations: YOLOv1 was implemented from scratch, Faster R-CNN and RetinaNet came from torchvision, YOLOv8 from Ultralytics, and DETR from HuggingFace Transformers. **No extensive hyperparameter tuning or architecture modifications** were performed beyond minor efficiency adjustments.

All training logs with per-epoch losses and validation metrics are saved under **results/training_processes.txt**, while YOLOv8 maintains its own log files inside the **results/yolov8/** folder. The trained weights for every model are stored in **results/models/**, allowing easy reuse for inference and further experiments. In addition, **training and validation curves** (losses and IoU trends) are available in **results/figures/**, providing a visual overview of how models converged during training and where overfitting or underfitting occurred.

Evaluation Metrics and Setup

To assess model performance, we used standard object detection metrics, computed on the test set only. These metrics capture both localization quality (how well predicted bounding boxes align with ground truth) and classification quality (whether the correct object class was predicted).

- **IoU@0.5 (Intersection over Union)** – The overlap ratio between predicted and ground-truth bounding boxes, counted as correct if IoU ≥ 0.5. This measures localization accuracy.

- **Precision@0.5** – Of all predicted boxes above the confidence threshold, the fraction that are true positives (predictions that correctly match a ground-truth object of the right class with IoU ≥ 0.5).
- **Recall@0.5** – Of all ground-truth objects in the dataset, the fraction that are successfully detected (matched by at least one predicted box of the correct class with IoU ≥ 0.5).
- **mAP@0.5 (mean Average Precision)** – The mean of Average Precision across all classes at IoU = 0.5. This is the most widely used benchmark for detection, as it balances precision and recall across different thresholds.
- **Training Time (s)** – Total wall-clock time spent training each model on our setup.
- **Average Inference Time (s)** – Mean per-image runtime during inference, measuring practical deployment speed.
- **#Params** – The number of trainable parameters in the model, reflecting model complexity and memory requirements.

These metrics together provide a balanced view: IoU, precision, recall, and mAP capture detection quality, while training/inference times and parameter counts measure efficiency and resource usage.

Results

The performance of all models on the test set is summarized in Table 1. The results highlight the trade-offs between accuracy, speed, and model size across different detection approaches.

Table 1: Performance comparison of all object detection models on the test set, reporting detection accuracy (IoU, Precision, Recall, mAP) alongside efficiency measures (training time, inference time, and parameter count).

Model	**IoU@0.5**	**Prec@0.5**	**Recall@0.5**	**mAP@0.5**	**Training Time (s)**	**Avg Inf. Time (s)**	**#Params**
YOLOv1	0.0023	0.0004	0.0011	-	1281.48	0.0074	24734367
Faster R-CNN	0.6160	0.2467	0.5431	0.3512	2550.36	0.0600	16003177
Retina Net	0.6765	0.1929	0.7579	0.3433	11663.87	0.3227	8776017
YOLOv8	0.5063	0.5278	0.3078	0.4127	16962.78	0.0353	2685343
DETR	0.2422	0.0172	0.1130	0.0073	1476.78	0.0284	18047754

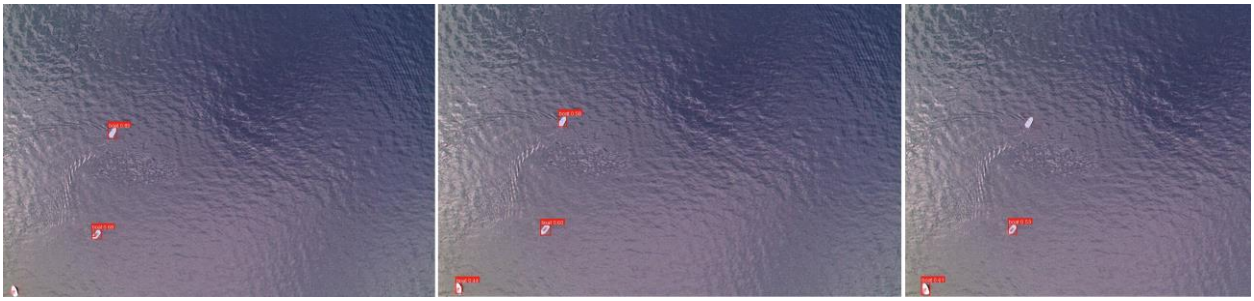
Key Findings

- **Overall performance.** RetinaNet achieved the highest IoU@0.5 (0.6765), showing strong recall but lower precision, while YOLOv8 balanced precision (0.528) and mAP@0.5 (0.413) best among all models. Faster R-CNN followed closely with solid IoU and recall, making it a reliable two-stage baseline. DETR struggled on this dataset, reflecting its sensitivity to small-object detection and limited data. YOLOv1 essentially failed, as expected from its older design.
- **Precision vs. Recall trade-off.** RetinaNet favored recall (0.758) at the cost of precision (0.193), while YOLOv8 showed the opposite trend, achieving the best precision but lower recall. Faster R-CNN offered a middle ground.
- **mAP@0.5 comparison.** YOLOv8 gave the highest mAP@0.5, confirming its strength on modern detection benchmarks, while RetinaNet and Faster R-CNN performed comparably. DETR's very low mAP confirmed it is less suited here without heavy tuning.
- **Training efficiency.** YOLOv1 trained fast but with useless results. DETR trained relatively quickly (1476s) but underperformed. RetinaNet and YOLOv8 required the longest training times (11.6k–16.9k seconds), reflecting their more complex pipelines.
- **Inference speed.** YOLOv1 was the fastest at inference (0.007s per image), followed by YOLOv8 (0.035s), which offers a realistic tradeoff between speed and accuracy. RetinaNet and Faster R-CNN were slower (0.32s and 0.06s), while DETR was also relatively slow.
- **Model complexity.** YOLOv1 had the highest parameter count (~24.7M) but delivered almost no performance. DETR (~18M) and Faster R-CNN (~16M) were heavy but not always efficient. YOLOv8 achieved the best parameter-efficiency tradeoff with only ~2.7M parameters, while still outperforming others in mAP.
- **Takeaway.** For small-object maritime detection, RetinaNet and YOLOv8 emerge as the most effective models: RetinaNet excels in recall (catching more objects), while YOLOv8 delivers higher precision and mAP with fewer parameters and faster inference. Faster R-CNN remains competitive, but DETR and YOLOv1 are not practical for this task.

Single Predictions

To better illustrate model behavior beyond aggregated metrics, we ran inference on images from the untouched original test set (which lacks annotations). Figure 8 shows predictions from three trained models, **Faster R-CNN, RetinaNet, and YOLOv8**, on the same image under different confidence thresholds.

Figure 8: Example predictions on the same test image. From left to right: (a) Faster R-CNN at threshold 0.6 (detects 2/3 boats), (b) RetinaNet at threshold 0.4 (detects all 3 boats), (c) YOLOv8 at threshold 0.4 (detects 2/3 boats).



At a threshold of **0.6**, Faster R-CNN successfully detects **2 out of 3 boats**, but misses the smallest target. RetinaNet, evaluated at a slightly lower threshold of **0.4**, detects **all 3 boats**, though at the cost of producing weaker confidence scores. YOLOv8, also at threshold **0.4**, detects **2 out of 3 boats**. This example highlights the tradeoff between **precision and recall** when adjusting detection thresholds: higher thresholds reduce false positives but risk missing small or low-visibility objects, while lower thresholds increase recall but can introduce more noise.

Figure 9: Single-image predictions at 0.6 threshold. From left to right: Faster R-CNN, RetinaNet, and YOLOv8. All three models successfully detect the main boats, with slightly different bounding box placements and confidence scores.

Figure 10: Single-image predictions at 0.4 threshold. From left to right: Faster R-CNN, RetinaNet, and YOLOv8. Faster R-CNN detects most swimmers and boats, while RetinaNet captures fewer but still reasonable targets. YOLOv8, despite being the most modern model, performs poorly here, failing to detect several obvious objects. This highlights that a newer or more advanced model does not always guarantee the best results on a given dataset.

Several more qualitative examples are provided in **results/test_images_preds/**.

Conclusion

This project aimed primarily at learning and practice rather than building production-ready detectors. The goal was to explore different object detection models on the challenging SeaDronesSee dataset and gain hands-on experience with both custom and pretrained implementations. We replicated the original YOLOv1 architecture (in a simplified form), while also leveraging pretrained models from **TorchVision** (Faster R-CNN, RetinaNet), **Hugging Face** (DETR), and ready-to-use models from **Ultralytics** (YOLOv8).

Through this process, we compared performance, efficiency, and complexity across one-stage and two-stage detectors. Results showed strong trade-offs: RetinaNet and Faster R-CNN performed well for small objects, YOLOv8 balanced speed with decent accuracy, and DETR highlighted the challenges of transformer-based detection without extensive tuning. Overall, the study deepened our understanding of how different detection strategies behave in a real-world maritime setting.

Future Work

- **Hyperparameter tuning** – Experiment with learning rates, schedulers, and data augmentation to unlock more performance, especially for DETR and YOLOv1.
- **Larger image sizes** – Re-train with higher resolutions (e.g., 640×640) to better capture very small objects at the cost of speed.
- **Model ensembling** – Combine predictions from multiple detectors to reduce false negatives.
- **Deployment focus** – Test lightweight variants (YOLOv8n, MobileNet backbones) for real-time drone or edge-device use.

References

[1] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection", *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR),* 2016.

[2] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

[3] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar, "Focal Loss for Dense Object Detection", *Proc. IEEE Int. Conf. Computer Vision (ICCV)*, 2017.

[4] Ultralytics, "YOLOv8: The Future of Vision AI", 2023.

[5] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, *"End-to-End Object Detection with Transformers"*, ECCV 2020.

Appendix - Implementation References

- **YOLOv1:** Pytorch YOLOv1 Tutorial (GitHub and YouTube walkthrough)
- **Faster R-CNN / RetinaNet:** Pytorch Tutorilas - TorchVision Object Detection Finetuning Tutorial and Simple Object Detection with Transfer Learning (plus a YouTube walkthrough)
- **YOLOv8:** Ultralytics Documentation and GitHub Repository
- **DETR:** Meta AI's official GitHub Repository and DETR-Factory-Pytorch