VIDEO ANALYTICS IN SEA ENVIRONMENT

Contributors	ID Number
Kanishk Yadav	2019B2A71562H
Shreyas Ravishankar Sheeranali	2019B3A70387P

BHARAT ELECTRONICS LIMITED, BENGALURU, INDIA

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI, INDIA

July 2021

ACKNOWLEDGEMENTS

We extend our sincere gratitude to the Practice School Division of BITS Pilani for providing us with the invaluable opportunity to undertake our Practice School-I program at Bharat Electronics Limited, Bangalore.

We are deeply thankful to Bharat Electronics Limited for welcoming us into their esteemed organization and allowing us to contribute to the development of a sea surveillance system.

Our heartfelt thanks go to Ms. Sravanthy for her support in facilitating the process and for assigning us a dedicated industry mentor. We are especially grateful to our project guide, Mr. Shrikant, for his active involvement in the project and for his continuous guidance and support throughout our internship.

We would also like to acknowledge the unwavering support of our PS Faculty, Dr. Raghunath Reddy M, whose commitment to the students and prompt assistance played a significant role in the smooth execution of our work.

ABSTRACT

The term *deep* in deep learning refers to the use of multiple layers within a neural network. The foundation of this field was laid in 1989, when George Cybenko published the first proof for sigmoid activation functions, which was later extended to multi-layer neural architectures by Kurt Hornik. Since then, deep learning has evolved rapidly.

As part of our internship at Bharat Electronics Limited, we set out to design a neural network architecture based on the YOLOv5 object detection model. The objective is to develop a model capable of accurately detecting and classifying ships in Indian waters.

This project report outlines the various deep learning concepts we explored through the four-course specialization on Coursera by Andrew Ng. Topics covered include the need for neural networks, hyperparameters, activation functions (ReLU, tanh, softmax, and sigmoid), fine-tuning, optimization, convolution, and regularization techniques (L2 and dropout).

The report also includes a brief overview of state-of-the-art object detection models—YOLOv3, YOLOv4, and YOLOv5. Finally, YOLOv5 has been implemented on our custom dataset using the Google Colab environment. A detailed account of the training and testing process, along with a comprehensive analysis of the results, is presented.

TABLE OF CONTENTS

GLOSSARY

1. **TensorFlow**: TensorFlow is a Python library for fast numerical computing created and released by Google. It is a foundation library that can be used to create Deep Learning models directly or by using wrapper libraries that simplify the process built on top of TensorFlow.
2. **Keras**: Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow with a focus on enabling fast experimentation.
3. **Matplotlib**: a plotting library for the Python programming language and its numerical mathematics extension NumPy.
4. **Accuracy**: The ratio between correctly predicted examples to the total number of examples
5. **Precision**: the ratio between the true positives and all the positives
6. **Recall**: the ratio between true positives and the sum of true positives and false negatives.
7. **Mean Average Precision** (mAP): the mean Average Precision over all classes and/or overall IoU thresholds, depending on the detection problem.
8. **Epochs**: the number of passes of the entire training dataset the machine learning algorithm has completed
9. **Image Augmentation**: Image augmentation are manipulations applied to images to create different versions of similar content to expose the model to a wider array of training examples.

INTRODUCTION

The Indian peninsula, bordered by the Indian Ocean to the south, the Bay of Bengal to the east, and the Arabian Sea to the west, requires advanced coastal surveillance systems to strengthen maritime security along its 7,520 km-long coastline.

Under the expert guidance of our mentors at Bharat Electronics Limited, we aim to develop an object detection and classification system for identifying ships in Indian waters. Upon successful implementation, this system will assist the Indian Coast Guard (ICG) in monitoring maritime activity with minimal human intervention.

At the core of our project lies the open-source YOLOv5 framework— a real-time object detection architecture developed by Ultralytics. Our approach also integrates foundational principles of Deep Learning and Neural Networks, including transfer learning and hyperparameter tuning, to optimize performance.

Key hyperparameters such as image size, number of epochs, batch size, and learning rate were adjusted iteratively to achieve optimal results. The outcomes of different training runs were visualized and analyzed using the wandb.ai platform. Code implementation made use of libraries such as TensorFlow, PyTorch, and Keras.

LEARNING FOR THE PROJECT

Course 1: Neural Networks and Deep Learning

We explored the significance of hidden layers in neural networks by comparing the performance of simple logistic regression with that of a network containing a single hidden layer using sigmoid activation.

It was observed that larger models, equipped with more hidden units, were able to fit the training data more effectively. However, beyond a certain point, excessively large models began to overfit the data, highlighting the need for balanced network design.

Course 2: Improving Deep Neural Networks – Hyperparameter Tuning, Regularization, and Optimization

Initialization

The initialization of parameters is as critical as training itself. To understand its impact, we applied three different types of parameter initialization on an elliptical two-class classification problem. All models were trained with the same number of iterations and identical hyperparameters. The comparison of their performances is illustrated in *Figure 1*.

Model	Train accuracy	Problem/Comment
3-layer NN with zeros initialization	50%	fails to break symmetry
3-layer NN with large random initialization	83%	too large weights
3-layer NN with He initialization	99%	recommended method

Regularization

L2 regularization incorporates a penalty term in the cost function by summing the squares of the weights. This discourages large weights, resulting in a smoother model where the output changes gradually with the input. The strength of this penalty is controlled by the hyperparameter *lambda*.

Dropout regularization randomly deactivates a fraction of neurons during each training iteration, with the dropout probability serving as the key hyperparameter.

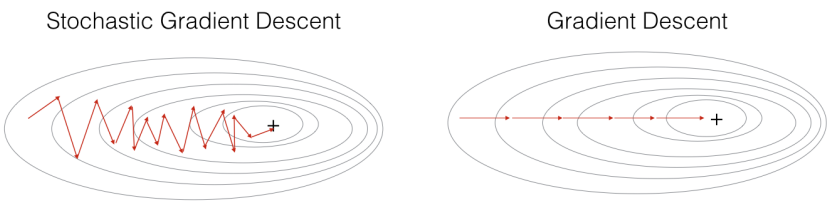
Despite a slight reduction in training performance, regularization methods like L2 and dropout help prevent overfitting. As shown in *Figure 2*, the regularized model performs better on unseen (test) data, compared to the overfitting observed in the non-regularized model.

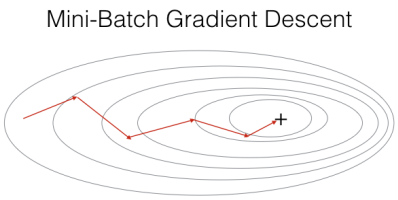
model	train accuracy	test accuracy
3-layer NN without regularization	95%	91.5%
3-layer NN with L2-regularization	94%	93%
3-layer NN with dropout	93%	95%

Gradient checking was also implemented to validate the correctness of backpropagation. This involves comparing gradients computed via backpropagation with those obtained through numerical approximation using forward propagation.

Optimization

We explored various optimization algorithms, including Stochastic Gradient Descent (SGD), Momentum, RMSProp, and Adam. Additionally, we utilized random mini batches to accelerate convergence and improve optimization efficiency. The behavioral differences among these optimizers are illustrated in *Figure 3*.





Performance comparisons indicate that the Adam optimizer converges significantly faster than mini-batch gradient descent and Momentum, although similar results can be achieved with more epochs using the latter methods. A summary of accuracy comparisons is provided in *Figure 4*.

optimization method	accuracy	cost shape
Gradient descent	>71%	smooth
Momentum	>71%	smooth
Adam	>94%	smoother

Lastly, learning rate decay was applied to enhance convergence stability, especially near the minimum of the cost function. The results with learning rate decay are summarized alongside optimization outcomes.

Course 3: Structuring Machine Learning Projects

Orthogonalization

Orthogonalization is a system design principle that ensures modifications to one component (such as an instruction or algorithm) do not introduce side effects in other components. This property simplifies testing and development by enabling independent verification of different parts of the system.

Precision and Recall

In situations where selecting the best classifier is difficult due to the precision-recall tradeoff, the F1 score provides a balanced evaluation. The F1 score is the harmonic mean of precision and recall:

$$F1\ score = 2\ /\ [(1\ /\ Precision) + (1\ /\ Recall)]$$

The goal is to maximize the F1 score.

Classifier	Precision	Recall	F1 Score
A	0.95	0.90	0.924
B	0.98	0.85	0.910

Train/Dev/Test Distributions

To ensure consistent evaluation, dev and test sets must originate from the same distribution. Setting up an appropriate dev set, and validation metric essentially defines the target the model aims to achieve.

In large-scale deep learning projects, it is common to allocate data as follows:

1. **98%** for training
2. **1%** for development
3. **1%** for testing

Comparison with Human-Level Performance

With recent advances, machine learning algorithms have become increasingly capable of approaching or even surpassing human-level performance in specific domains. Once human-level performance is reached, further improvements become difficult due to the proximity to the **Bayes-optimal error**—the irreducible minimum error rate.

Surpassing human performance is especially challenging in tasks requiring natural perception. However, some applications where machines already outperform humans include:

1. Online advertising
2. Product recommendation
3. Loan approval

Improving Model Performance

Error Analysis

Deep learning models are generally robust to random errors in training data but are more sensitive to systematic labeling errors. To address mislabeled data in the dev/test sets, performing error analysis is recommended.

Handling High Variance

In the presence of high variance (i.e., overfitting), several strategies can be applied:

1. Acquiring more training data
2. Applying regularization techniques (L2, Dropout, Data Augmentation)
3. Searching for a more suitable neural network architecture or optimizing hyperparameters

Course 4: Convolutional Neural Networks

Convolution is a fundamental operation in image processing, designed to extract meaningful features from input images. It works by performing a dot product between the input data and a set of 2D filters (also known as kernels or weight matrices). In a convolutional layer, these filters slide over the input image, transforming the input volume (height, width, and color channels) into an output volume of different dimensions, depending on the number and size of filters used.

Padding is employed to preserve spatial dimensions (height and width) after convolution. This enables the construction of deeper networks without losing information at the edges of the input.

Pooling layers are used to progressively reduce the spatial dimensions of the feature maps. By applying operations such as max pooling or average pooling within a sliding window, these layers help in down-sampling the input while retaining important features, thus reducing computational load and overfitting.

Car Detection with YOLO

We applied the YOLOv1 (You Only Look Once) algorithm for object detection on images of resolution 608 × 608 pixels, covering 80 different classes and utilizing 5 anchor boxes. The YOLO encoding enabled us to localize and classify multiple objects within a single image efficiently.

To refine the predictions, **non-max suppression** was implemented to eliminate bounding boxes with low confidence scores and high Intersection over Union (IoU) with other boxes, thereby avoiding duplicate detections of the same object. From the processed output, the top 10 bounding boxes were selected as the final predictions for test set images.

```
out_scores, out_boxes, out_classes = predict("test.jpg")
```

```
Found 10 boxes for images/test.jpg
car 0.89 (367, 300) (745, 648)
car 0.80 (761, 282) (942, 412)
car 0.74 (159, 303) (346, 440)
car 0.70 (947, 324) (1280, 705)
bus 0.67 (5, 266) (220, 407)
car 0.66 (706, 279) (786, 350)
car 0.60 (925, 285) (1045, 374)
car 0.44 (336, 296) (378, 335)
car 0.37 (965, 273) (1022, 292)
traffic light 0.36 (681, 195) (692, 214)
```

Course 5: Python for Computer Vision using OpenCV and Deep Learning

This course served as an introduction to image processing using OpenCV, covering fundamental manipulation techniques such as smoothing, blurring, thresholding, and morphological operations. These techniques, combined with libraries like OpenCV and NumPy, make object detection tasks—such as identifying edges, corners, and grids—more efficient and accessible.

Face Detection Using Haar Cascades in OpenCV

We implemented face detection using Haar Cascade classifiers on the well-known Solvay Conference photograph. Prior to running the detection algorithm, the image was converted to grayscale, a necessary preprocessing step for Haar cascades to function correctly.

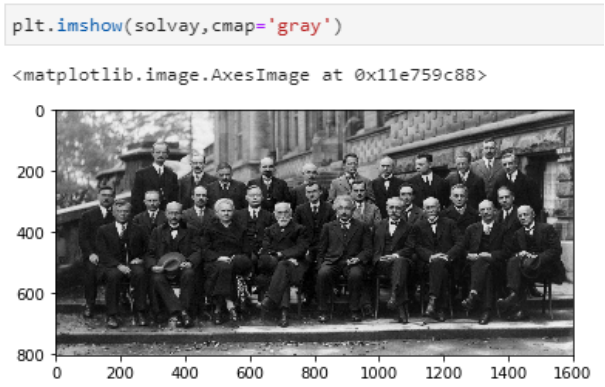


Figure 6. The Solvay Conference

Results

The Haar Cascade classifier successfully identified multiple faces in the image.

```
result = detect_face(solvay)
plt.imshow(result,cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x11fff0198>
```

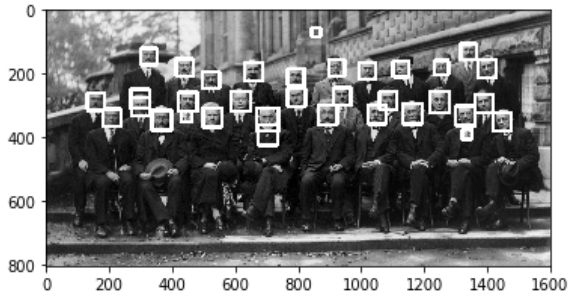


Figure 7. Detected Faces

Improving Accuracy by Avoiding Side Face Detection

To enhance detection accuracy and reduce false positives, particularly for side profiles, we adjusted key parameters in the detection algorithm such as *minNeighbors* and *scaleFactor*. These modifications refined the results, focusing the detection on frontal faces.

```
def adj_detect_face(img):

    face_img = img.copy()

    face_rects = face_cascade.detectMultiScale(face_img,scaleFactor=1.2, minNeighbors=5)

    for (x,y,w,h) in face_rects:
        cv2.rectangle(face_img, (x,y), (x+w,y+h), (255,255,255), 10)

    return face_img
```

```
result = adj_detect_face(solvay)
plt.imshow(result,cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x11ffc89e8>
```

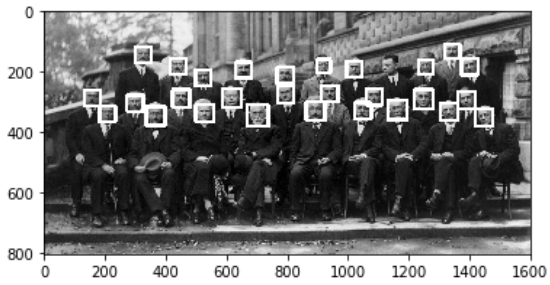


Figure 8. Optimized Detection Output

THE YOLO EVOLUTION

Object classification helps identify the contents of an image, while object detection determines the locations of multiple objects within that image. Early detection systems repurposed object classifiers to perform detection, using the sliding windows technique, which evaluates a classifier at various locations and scales within a test image. More advanced methods like R-CNN (Region-Based Convolutional Neural Networks) use region proposal techniques to generate potential bounding boxes in an image, followed by applying classifiers on these proposed regions. However, these complex pipelines are slow and difficult to optimize, as each component requires independent training.

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi introduced **"You Only Look Once: Unified, Real-Time Object Detection",** which presented a revolutionary approach to object detection. YOLO (You Only Look Once) redefines object detection as a single regression problem, directly mapping image pixels to bounding box coordinates and class probabilities. This approach enables a single convolutional network to predict multiple bounding boxes and their respective class probabilities simultaneously.

YOLOv1

YOLO offers several advantages over its predecessors, notably its simplicity and speed. Unlike sliding window or region proposal-based techniques, YOLO processes the entire image during both training and testing, which allows it to implicitly learn class information and object appearance.

The architecture of YOLOv1, inspired by GoogLeNet, comprises 24 convolutional layers followed by two fully connected layers. Instead of GoogLeNet's inception modules, YOLO uses 1x1 reduction layers followed by 3x3 convolutional layers.

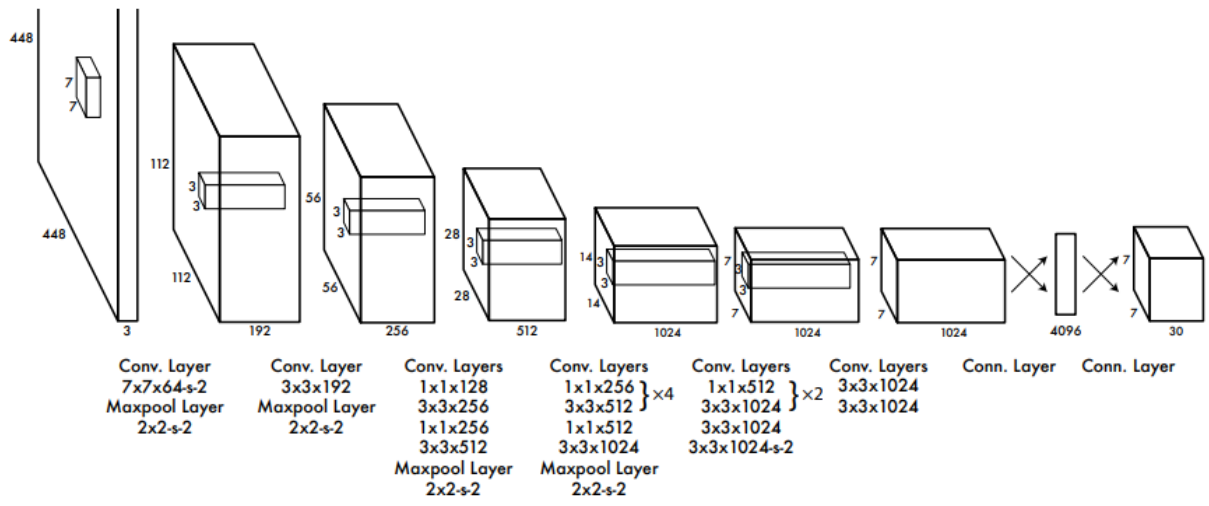


Figure 9. Architecture of YOLOv1

Despite its effectiveness, YOLOv1 has limitations. A single grid cell can only predict two bounding boxes and one class, which limits the number of potential predictions. Additionally, the loss function in YOLOv1 is sensitive to object scale, resulting in disproportionately smaller loss values for larger objects.

YOLOv2

To address the shortcomings of YOLOv1, Redmon and Farhadi introduced **YOLOv2**, incorporating several architectural improvements. Key advancements include:

1. High-resolution classifiers
2. Convolution with anchor boxes
3. Dimension clusters
4. Direct location prediction
5. Fine-grained feature extraction
6. Multi-scale training

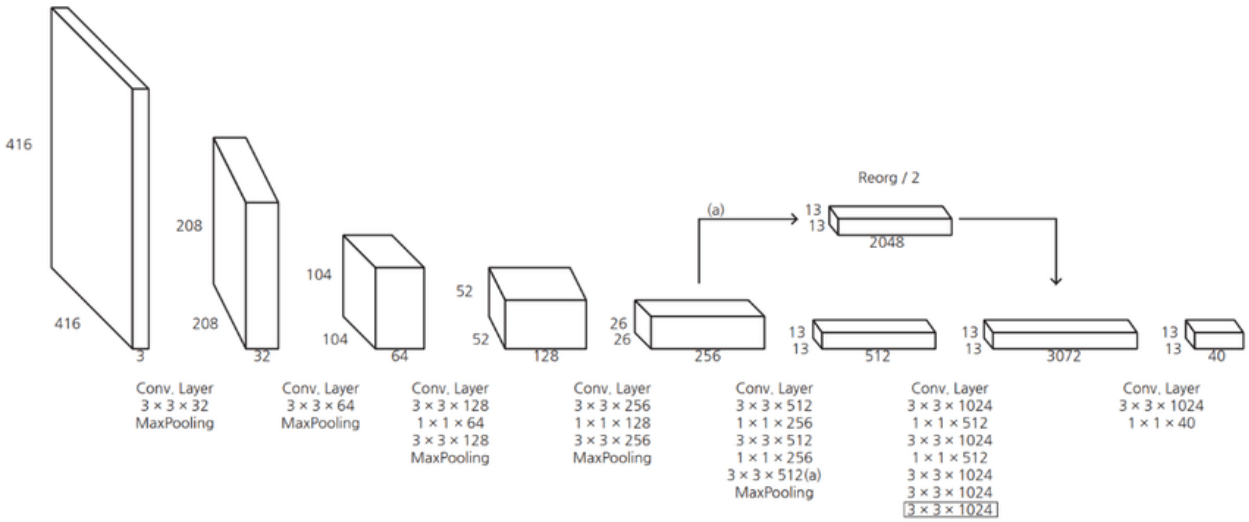


Figure 10. Architecture of YOLOv2

YOLOv3

YOLOv3, developed by Redmon and Farhadi, further refines YOLO's approach. It runs at 22 ms for 320x320 resolution images with 28.2 mAP (mean average precision), making it three times faster than SSD (Single Shot Multibox Detector). YOLOv3 improves object detection by using multiple scales and the Darknet-53 architecture, which enhances the model's ability to handle small objects while improving computational efficiency.

Bounding box prediction in YOLOv3 involves predicting four coordinates for each bounding box, and the network uses logistic regression to assign an objectness score.

$$b_x = \sigma(t_x) + c_x$$
$$b_y = \sigma(t_y) + c_y$$
$$b_w = p_w e^{t_w}$$
$$b_h = p_h e^{t_h}$$

Figure 11. Mathematical Relations for Bounding Boxes

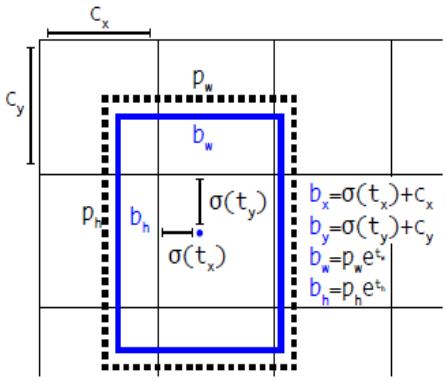


Figure 12. Bounding Boxes Visualization

YOLOv3 predicts boxes at three different scales, using pyramid networks to extract features, with independent logistic classifiers and binary cross-entropy loss for class predictions.

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 13. Darknet-53 Architecture

YOLOv3's multi-scale predictions enable improved performance on smaller objects but exhibit slightly reduced accuracy for medium and large objects.

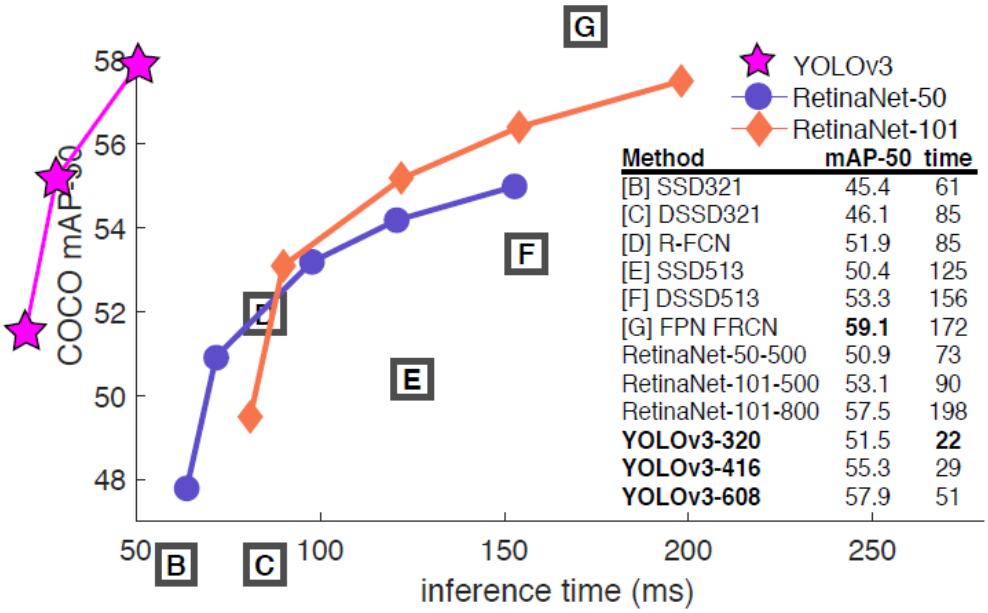


Figure 14. Performance Comparison of YOLOv3, RetinaNet-50, and RetinaNet-101

YOLOv4

YOLOv4, a CNN-based object detection model, addresses the issue of real-time operation on conventional GPUs. It introduces several features to improve performance, such as Weighted-Residual Connections (WRC), Cross-Stage Partial Connections (CSP), and DropBlock regularization. YOLOv4 achieves 43.5% AP on the MS COCO dataset at real-time speeds of 65 FPS on Tesla V100 GPUs.

A modern object detection model typically consists of:

1. **Backbone:** Pre-trained on datasets like ImageNet (e.g., VGG, ResNet, DenseNet)
2. **Head:** Responsible for predicting the classes and bounding boxes of objects
3. **Neck:** Collects feature maps from different stages and facilitates feature aggregation (e.g., Feature Pyramid Networks).

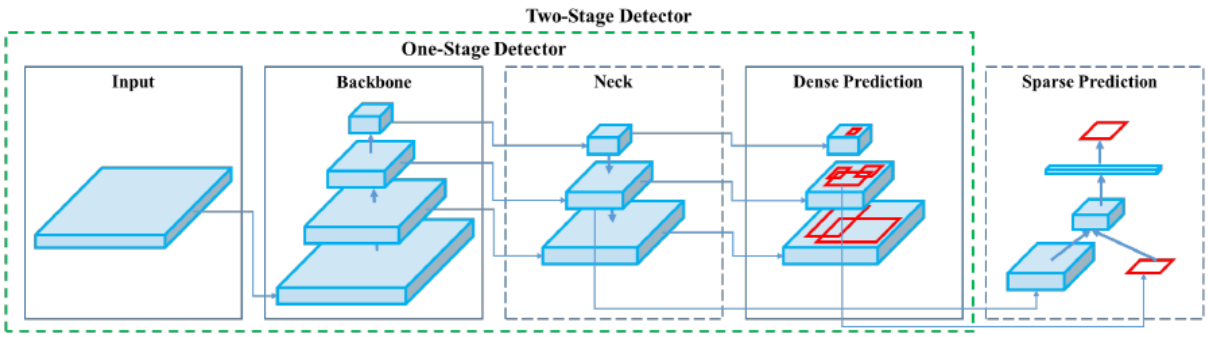


Figure 15. Components of a Modern Object Detector

Architecture Selection for YOLOv4

The objective in architecture selection is to achieve a balance between input network resolution, layer depth, parameter count, and the output layer. The aim is to optimize the model for detecting objects of different sizes, requiring higher network resolution, more layers for increased receptive field, and additional parameters for detecting various object sizes.

The **Bag of Freebies** *(BoF)* and **Bag of Specials** *(BoS)* approaches are used for enhancing object detection training, incorporating features like various activation functions, bounding box regression losses, and data augmentation techniques.

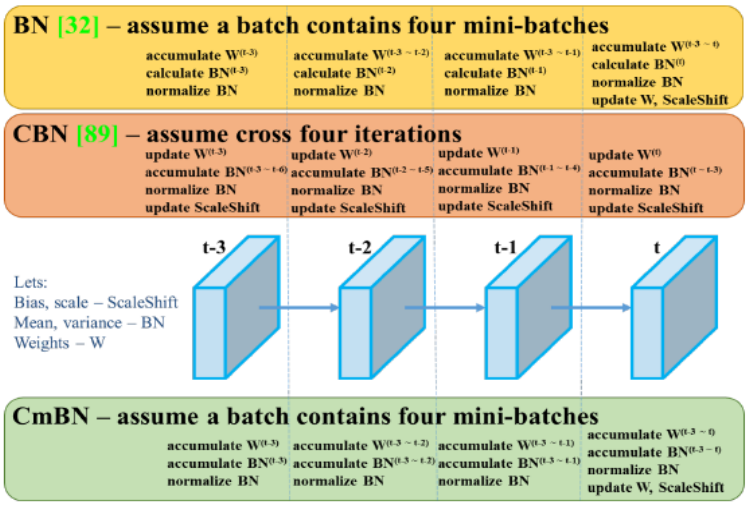


Figure 16. Normalization Techniques in YOLOv4

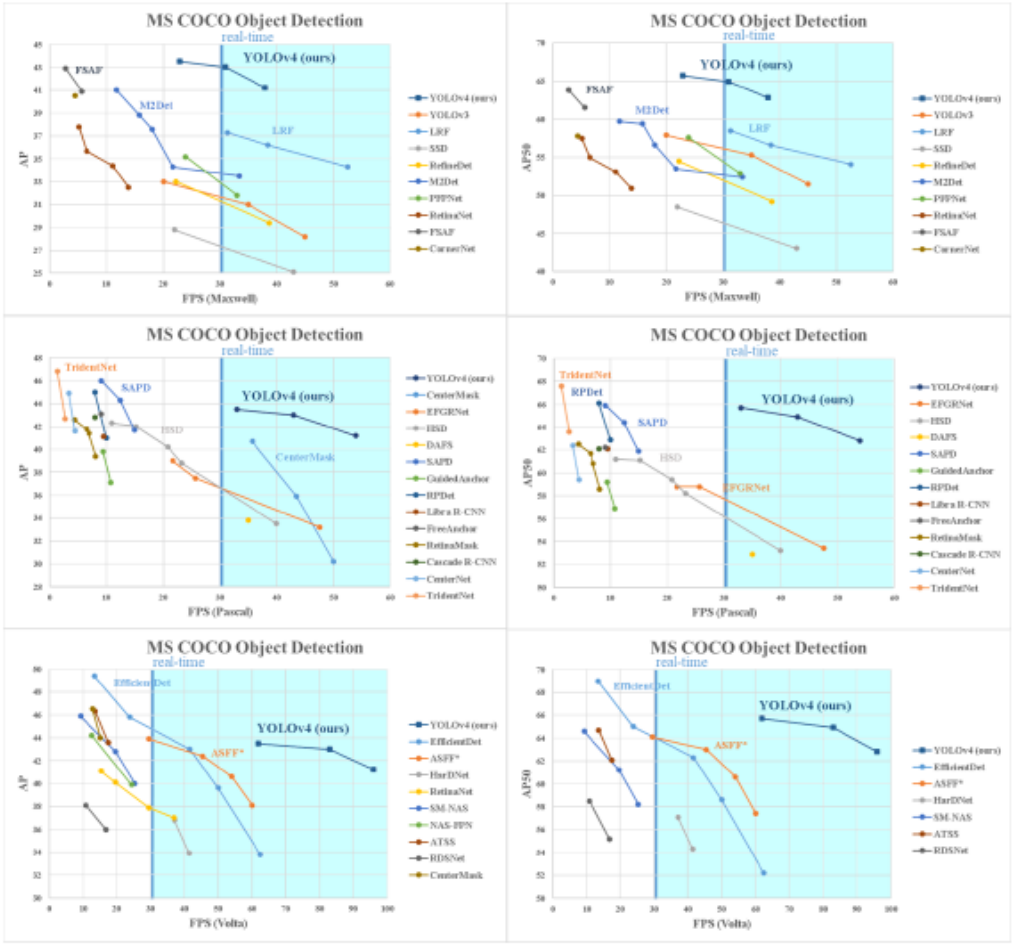


Figure 17. Performance Graphs on COCO Dataset

YOLOv4 offers a state-of-the-art, fast, and accurate detector capable of training on conventional GPUs, with the added advantage of being optimized for real-time performance.

YOLOv5

YOLOv5, developed by Glenn Jocher from Ultralytics, is an extension of the YOLOv3 PyTorch repository. It retains the essential structure of YOLOv3, with three primary components:

1. **Backbone:** Extracts key features using CSPNet, which improves processing speed in deeper networks.
2. **Neck:** Generates feature pyramids, aiding in generalization and object detection across various scales.
3. **Head:** Responsible for applying anchor boxes and generating final output vectors with class probabilities, objectness scores, and bounding boxes.

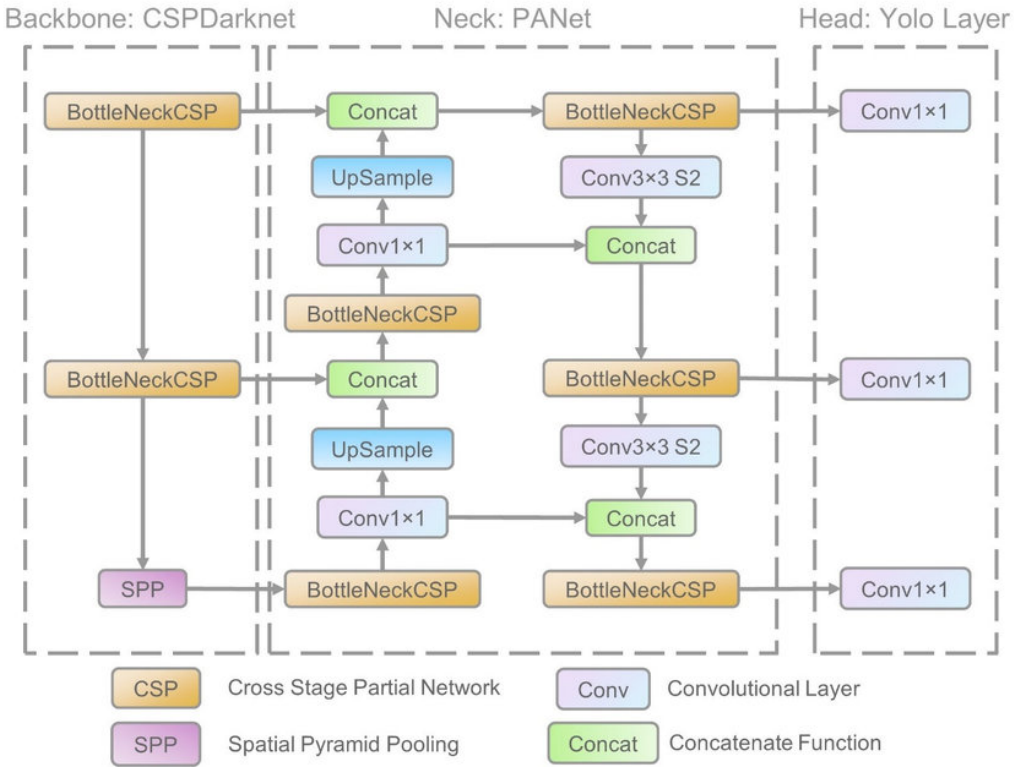


Figure 18. Architecture of YOLOv5

With these enhancements, YOLOv5 offers improved efficiency and scalability, making it a valuable tool for real-time object detection applications.

IMPLEMENTING YOLOv5

Training

The dataset consisted of **6350 images** labeled with bounding box coordinates and ship classes. The images were divided into a 70:30 ratio for training and test sets, with **4470 training examples** and **1880 test examples**. The dataset contained **nine** classes of ships: [Ferry, Buoy, Vessel/ship, Speed boat, Boat, Kayak, Sailboat, Flying bird/plane, Other]

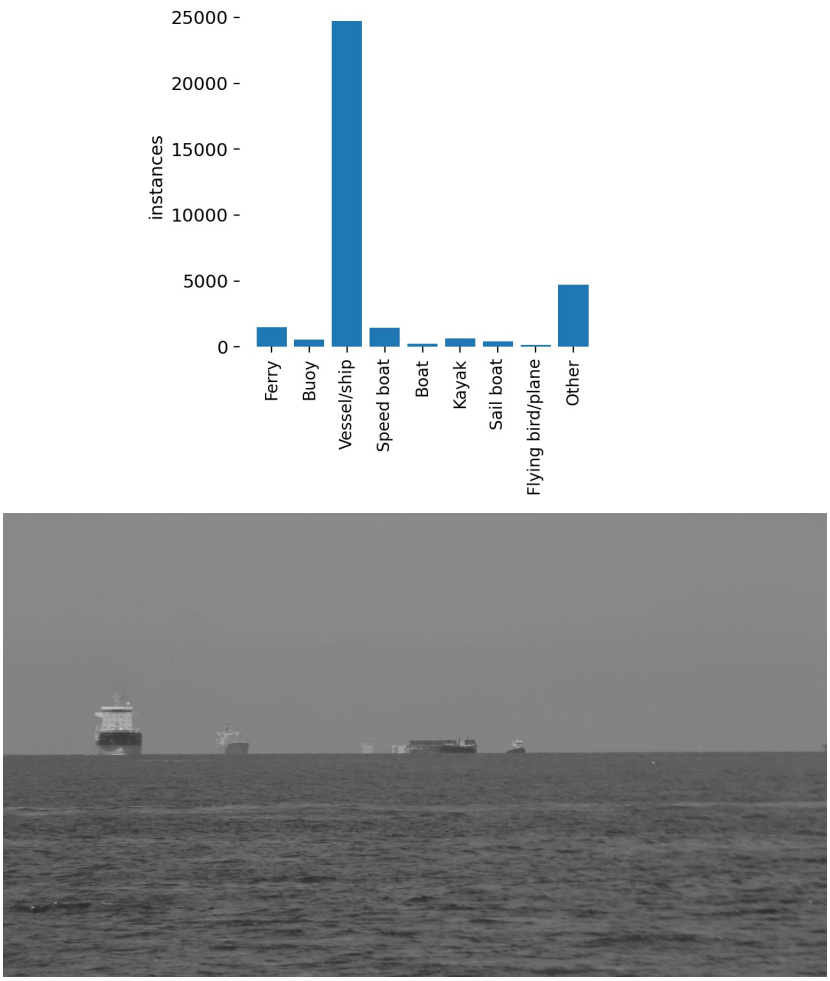


Figure 19. shows the distribution of class instances in the training set and a representative image from the dataset.

After loading the YOLOv5 repository from GitHub, the default hyperparameters were used for the first iteration: **batch size of 16**, **image size of 640**, and **5 epochs**. The **ships.yaml** file specified the locations of training and test images and labels, along with the number of classes. YOLOv5 **v5s variant** (283 layers, 70,85,118 parameters) was used, and **transfer learning** was applied by initializing weights with **yolov5s.pt**, pre-trained on the **COCO** dataset.

The following command was used to start training:

```
!python train.py --img 640 --batch 16 --epochs 5 --data ../ships.yaml --weights yolov5s.pt
```

The results after this iteration were:

```
      Epoch   gpu_mem       box       obj       cls     total    labels  img_size
        4/4     3.58G   0.05098   0.03693   0.02211      0.11       120       640: 100% 1
                Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95:
                  all       4470      34376      0.908      0.133      0.162     0.0728
                Ferry       4470       1482          1          0     0.0295    0.00909
                 Buoy       4470        531          1          0     0.0649     0.0306
          Vessel/ship       4470      24746      0.523      0.729       0.61      0.249
           Speed boat       4470       1469          1          0     0.0218    0.00672
                 Boat       4470        226          1          0    0.00592    0.00122
                Kayak       4470        633          1          0    0.00105   0.000294
            Sail boat       4470        433      0.648      0.463      0.593      0.327
  Flying bird/plane       4470        132          1          0          0          0
                Other       4470       4724          1          0      0.135     0.0316
5 epochs completed in 0.810 hours.
```

Figure 20: Final output from GColab after training for five epochs with initial weights (yolov5s.pt)

Performance metrics for all classes were:

1. Precision: **0.908**
2. Recall: **0.133**
3. Mean Average Precision @ 0.5 (mAP @ 0.5): **0.162**
4. Mean Average Precision @ 0.95 (mAP @ 0.95): **0.0728**

The **0.5 threshold** for mAP means objects with >50% overlap between predicted and ground truth bounding boxes are considered positive.

As typical object detection models train thousands of epochs at once, the training was constrained to **Google Colab**, limiting GPU access. As a result, training was split into multiple iterations of **5-10 epochs**. The weights were stored after each iteration and reapplied for the next. The following command was used to resume training:

!python train.py --img 640 --batch 16 --epochs 5 --data ../ships.yaml --weights ../last.pt

This iterative process continued until **120 cumulative epochs** were reached, and the results are summarized in **Table 1**.

Event	Image size	Batch	Epochs	Total epochs	Precision	Recall	F1	mAP @ 0.5	mAP @ 0.95
7/8/21 12:07	640	16	5	5	0.74	0.403	0.5218	0.445	0.24
7/8/21 17:41	640	16	5	10	0.754	0.582	0.6569	0.602	0.313
7/8/21 20:51	640	16	10	20	0.895	0.728	0.8029	0.769	0.453
7/8/21 22:41	640	16	10	30	0.925	0.746	0.8259	0.775	0.471
7/9/21 9:02	640	16	10	40	0.914	0.739	0.8172	0.769	0.478
7/9/21 11:37	640	16	10	50	0.918	0.748	0.8243	0.775	0.483
7/9/21 14:56	640	16	10	60	0.919	0.746	0.8235	0.774	0.489
7/9/21 22:17	640	16	20	80	0.93	0.76	0.8364	0.813	0.518
7/10/21 18:10	640	32	20	100	0.912	0.788	0.8455	0.804	0.525
7/11/21 14:26	640	32	20	120	0.947	0.79	0.8614	0.817	0.537

As observed, the improvement in metrics slowed down significantly after 40 epochs. Given that the original hyperparameters were designed for the **COCO dataset**, adjustments were needed for better performance.

TUNING HYPERPARAMETERS

Observation I

Learning Rate

Training with small epoch batches (5-10 epochs) without learning rate decay resulted in a high initial learning rate. To address this, the learning rate was manually decreased with cumulative epochs. A comparison was made between two learning rates (0.001 and 0.005) for 10 epochs, with the following results:

Figure 21: Training loss (class, box, and object) and mAP @0.95 for learning rates 0.001 (Red) and 0.005 (Green)

The lower learning rate (0.001) resulted in smoother convergence and lower training loss, achieving a higher mAP.

Observation II

Image Size

The image size was set to **640** initially, but given the dataset's image resolution of **1920x1080**, it led to a loss of fine details when resized. Due to GPU RAM limitations on Colab, an image size of **960** was tested, resulting in better performance. With 30 cumulative epochs, **mAP @ 0.5** was greater than **0.8**, and **mAP @ 0.95** exceeded **0.5**.

Figure 22. A sample mosaic generated from the training set

A retraining was performed from scratch with the following new hyperparameters, yielding the results in **Table 2**.

Event	Image size	Batch	Epochs	Learning Rate	Total epochs	Precision	Recall	mAP @0.5	mAP @ 0.95	F1
7/12/21 18:50	960	32	10	0.01	10	0.852	0.676	0.689	0.389	0.7539
7/13/21 11:43	960	16	10	0.01	20	0.917	0.773	0.813	0.495	0.8389
7/13/21 13:30	960	16	10	0.005	30	0.949	0.777	0.827	0.51	0.8544
7/13/21 18:19	960	16	10	0.005	40	0.921	0.807	0.849	0.509	0.8602
7/13/21 19:52	960	16	10	0.001	40	0.941	0.779	0.827	0.518	0.8524
7/14/21 14:35	960	32	10	0.001	58	0.953	0.787	0.846	0.537	0.8621
7/14/21 19:09	960	32	10	0.001	70	0.947	0.788	0.854	0.532	0.8602
7/15/21 12:14	960	32	10	0.0005	80	0.957	0.783	0.851	0.546	0.8613
7/17/21 12:59	960	32	10	0.0005	90	0.903	0.821	0.859	0.551	0.8600
7/17/21 22:45	960	32	10	0.0001	100	0.907	0.826	0.862	0.552	0.8646

As shown, the **modified hyperparameters** (image size of **960** and gradual learning rate decay) resulted in significantly improved performance in fewer epochs.

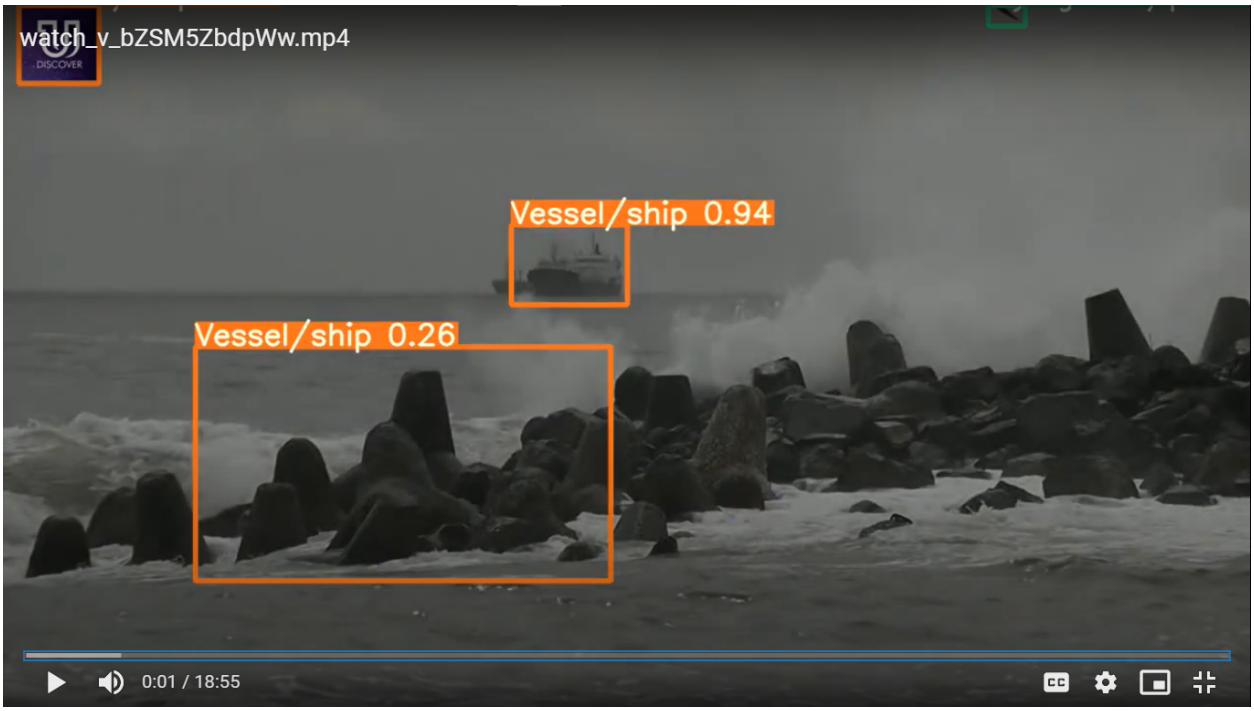
CROSS VALIDATION: YOUTUBE

Finally, to check how our model generalizes with the real world, we applied our parameters (from the second model, trained after modifying the hyperparameters) on two **YouTube** videos. We took all screenshots at a constant resolution of 720p, but the image clarity depends on the clip in the YouTube video.

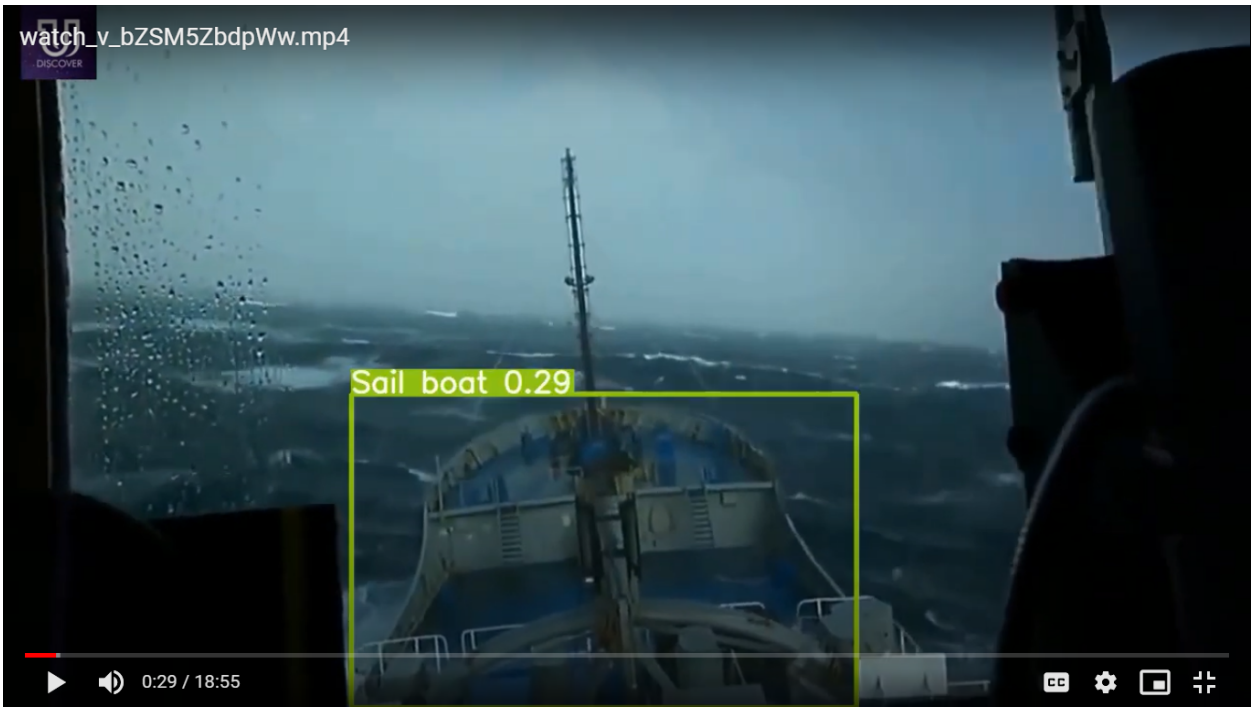
Video 1: Ships in Horrible Storms

Code for execution:

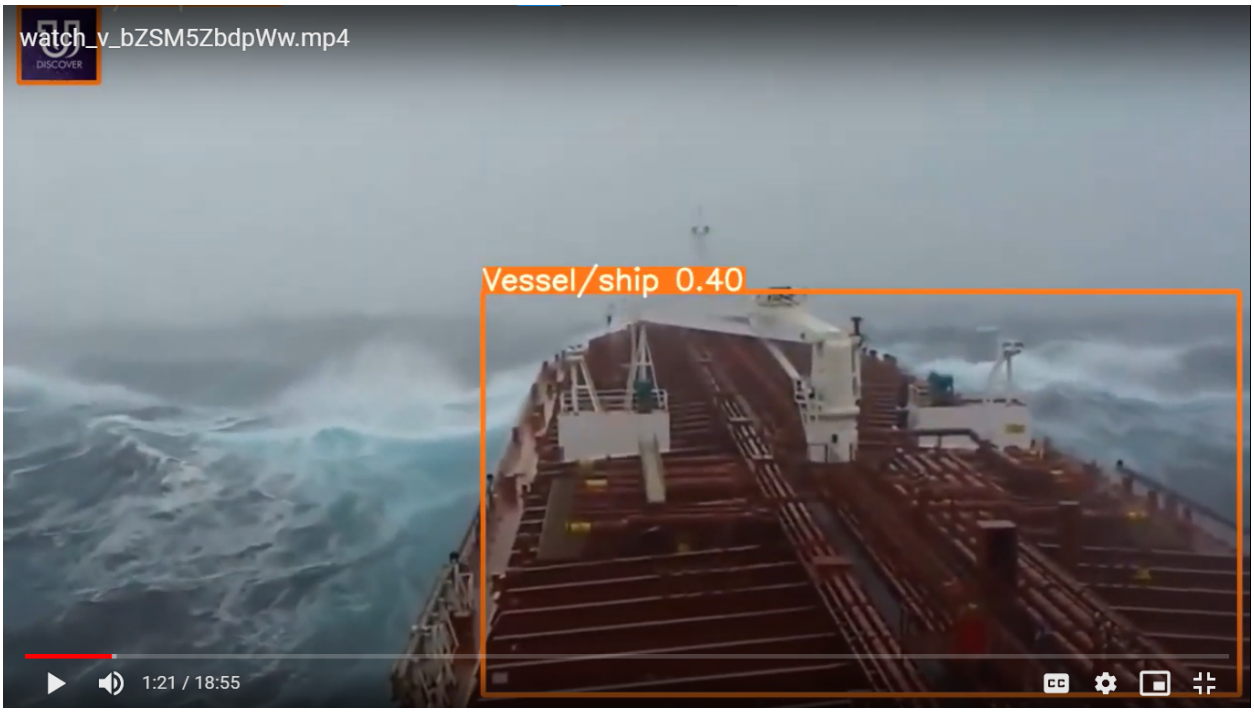
!python detect.py --source 'https://www.youtube.com/watch?v=bZSM5ZbdpWw' --weights ../17_7_21_22HR/weights/last.pt --img 960



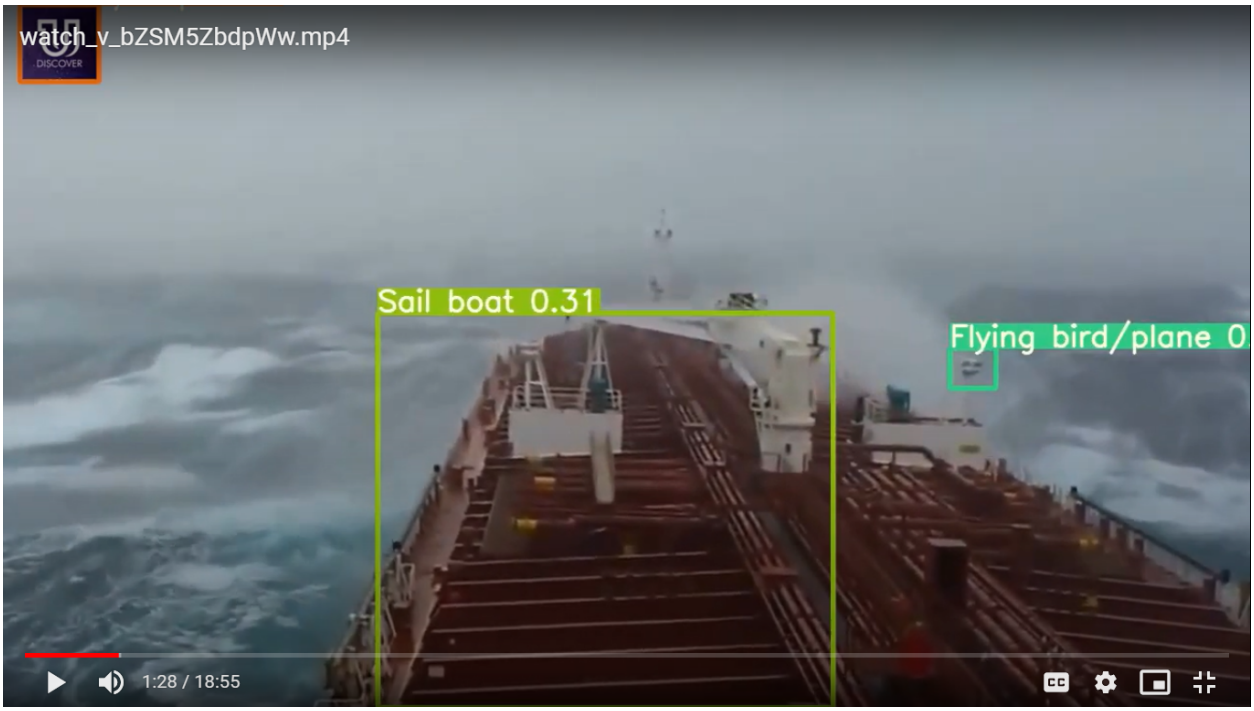
Detect Image 1. False detection of rock as vessel/ship, and correct identification of a distant ship.



Detect Image 2. Part of a vessel/ship wrongly classified as a sailboat.



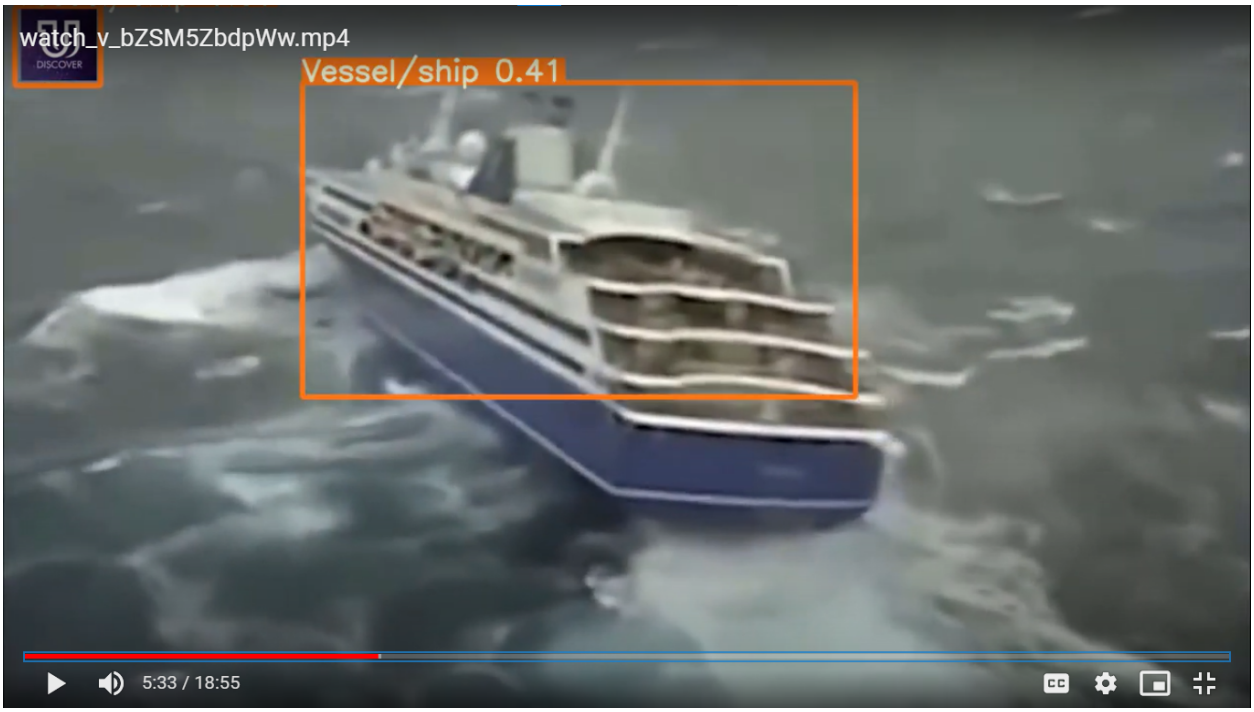
Detect Image 3. Part of a vessel/ship correctly classified.



Detect Image 4. Image like previous (3) has false positives in Sail boat and flying bird/plane class.

Detect Image 5. False positive recognition of sailboat

Detect Image 6. Correct detection of a buoy in a low-resolution image.



Detect Image 7. Correct detection of a ship in an inclined orientation.

Detect Image 8. Correct detection of a speed boat in a low-resolution image with surrounding noise.

Detect Image 9. Correct detection of a ship in a low-resolution image in horizontal orientation.

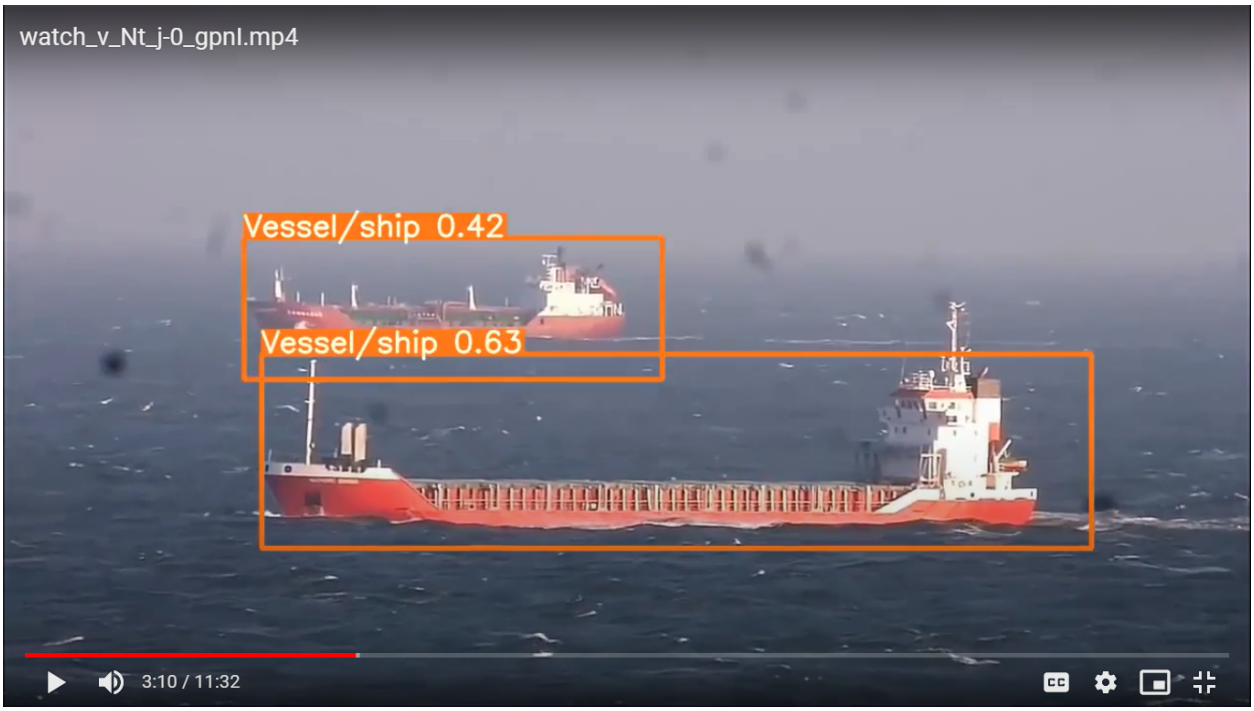
Detect Image 10. Correct detection of a sailboat in a clear image.

Video 2: Types of ships in Merchant navy

Code for execution:

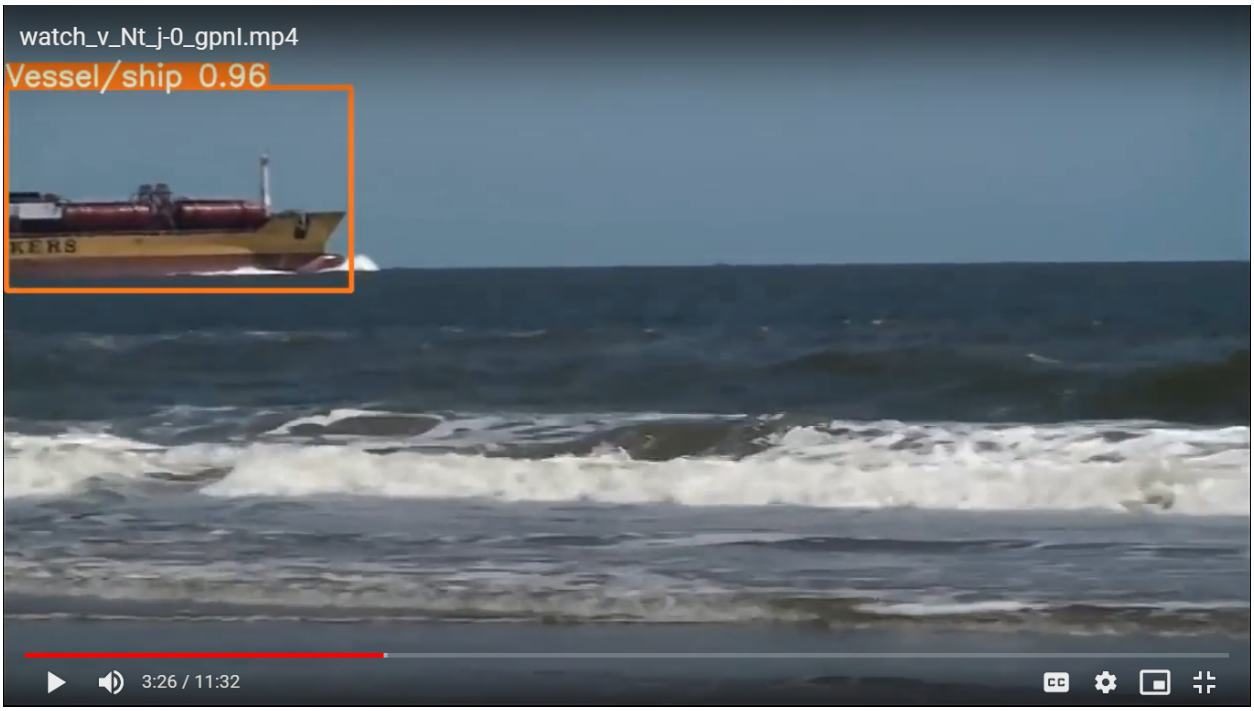
!python detect.py --source 'https://www.youtube.com/watch?v=Nt_j-0_gpnI' --weights ../17_7_21_22HR/weights/last.pt --img 960

Detect Image 11. Correct identification of a ship/vessel, image similar to dataset images.

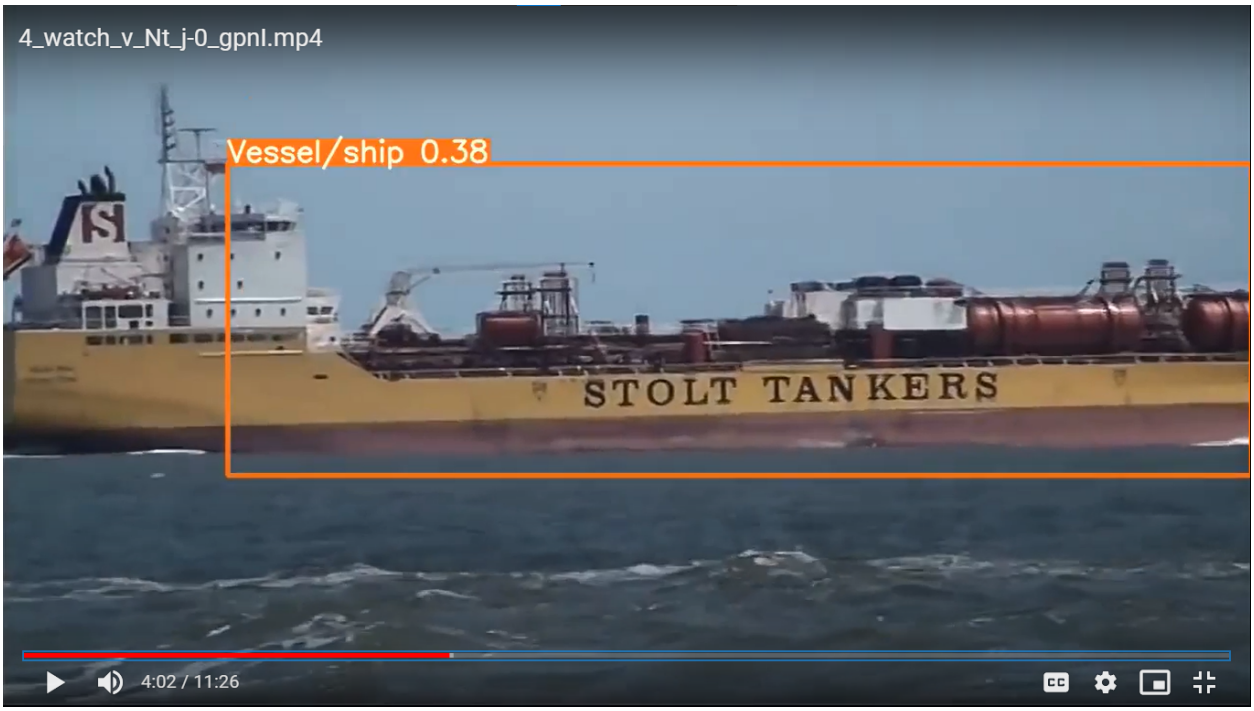


Detect Image 12: Correct identification of two instances of ship/vessel, image similar to dataset images.

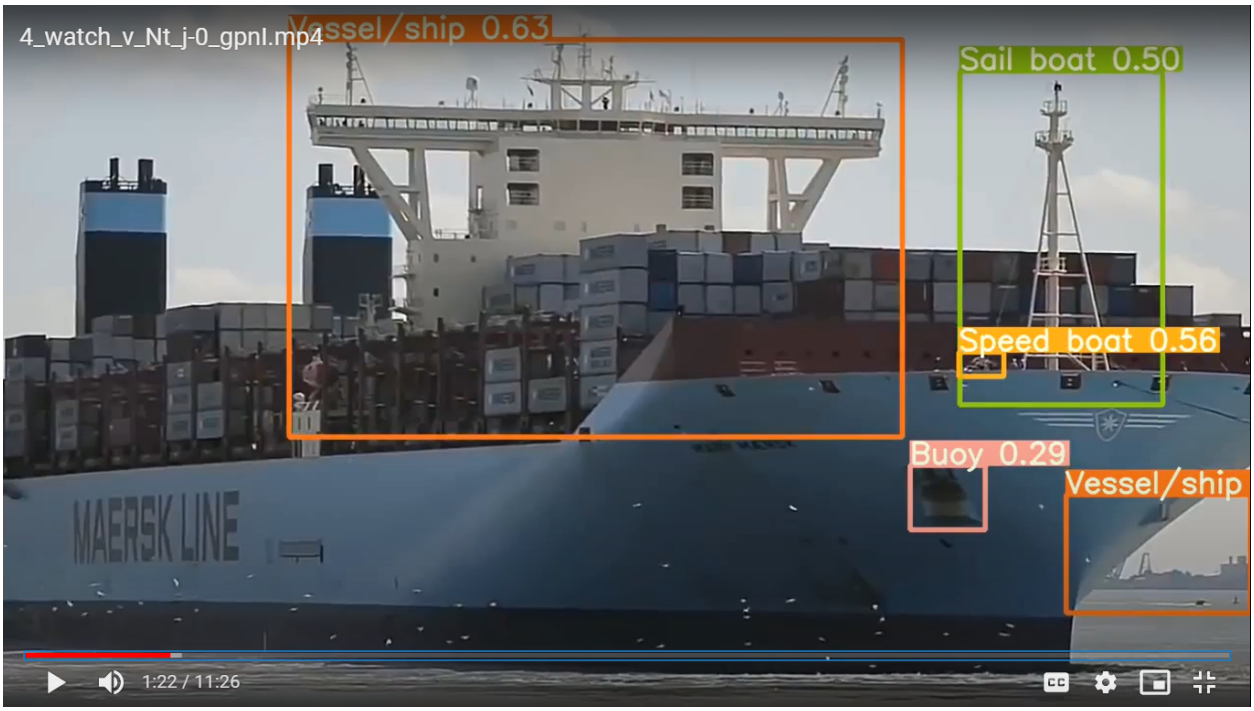
Detect Image 13: The approaching vessel is incorrectly identified as a speed boat.



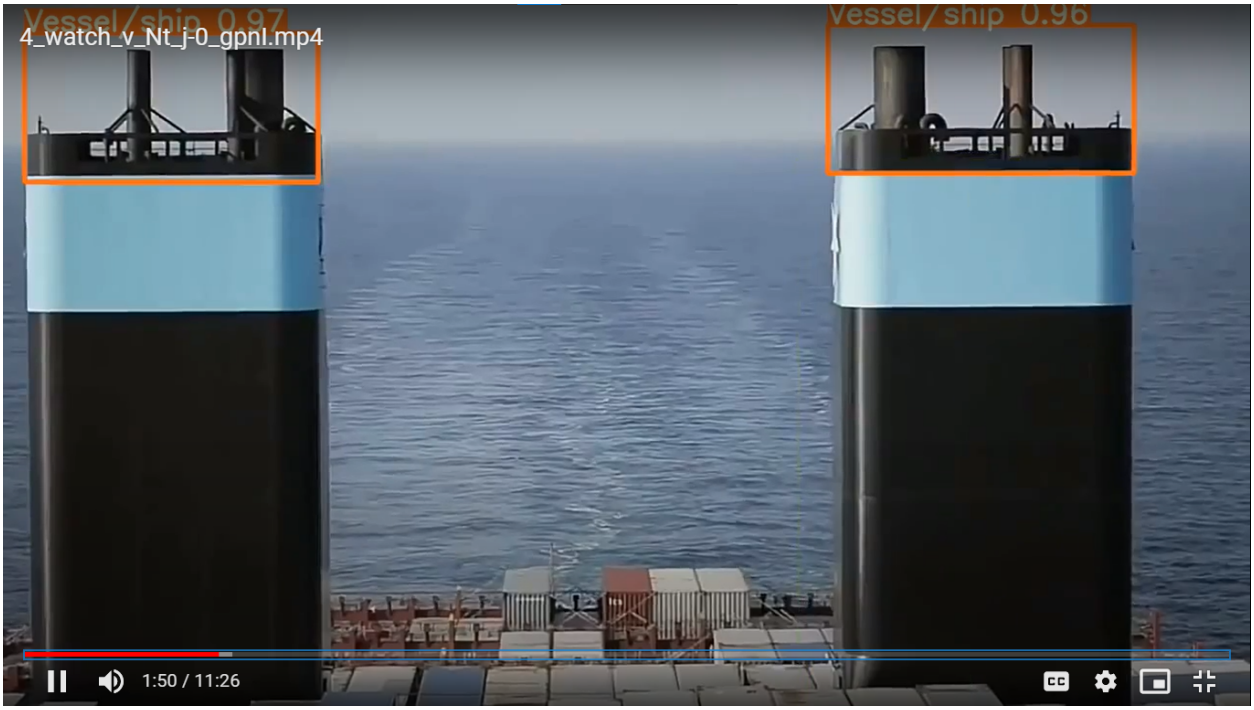
Detect Image 14: Vessel correctly identified with 0.96 confidence after more of it was visible.



Detect Image 15: Confidence decreases when the complete ship/vessel is not visible.

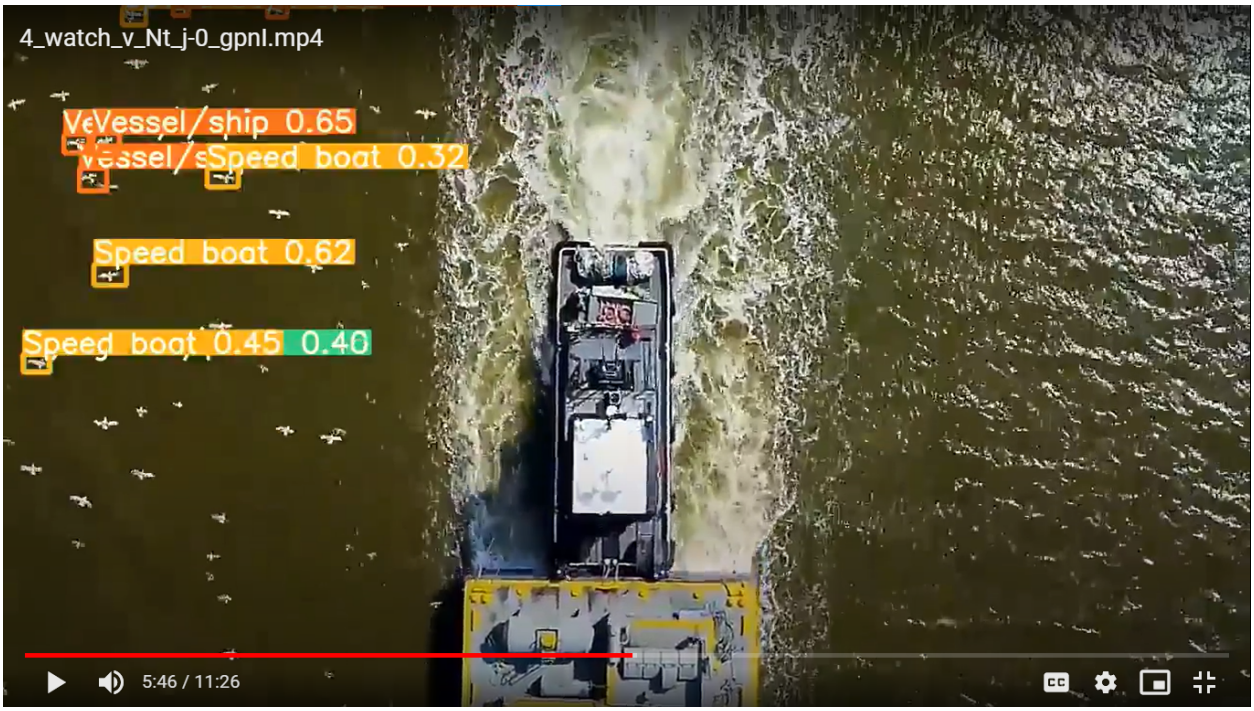


Detect Image 16: Correctly identifies a distant ship/vessel (bottom right of the image) vessel/ship Multiple false positives: Sailboat (mislead by the sailboat mast shaped structure), buoy and (structures on the surface of the ship).

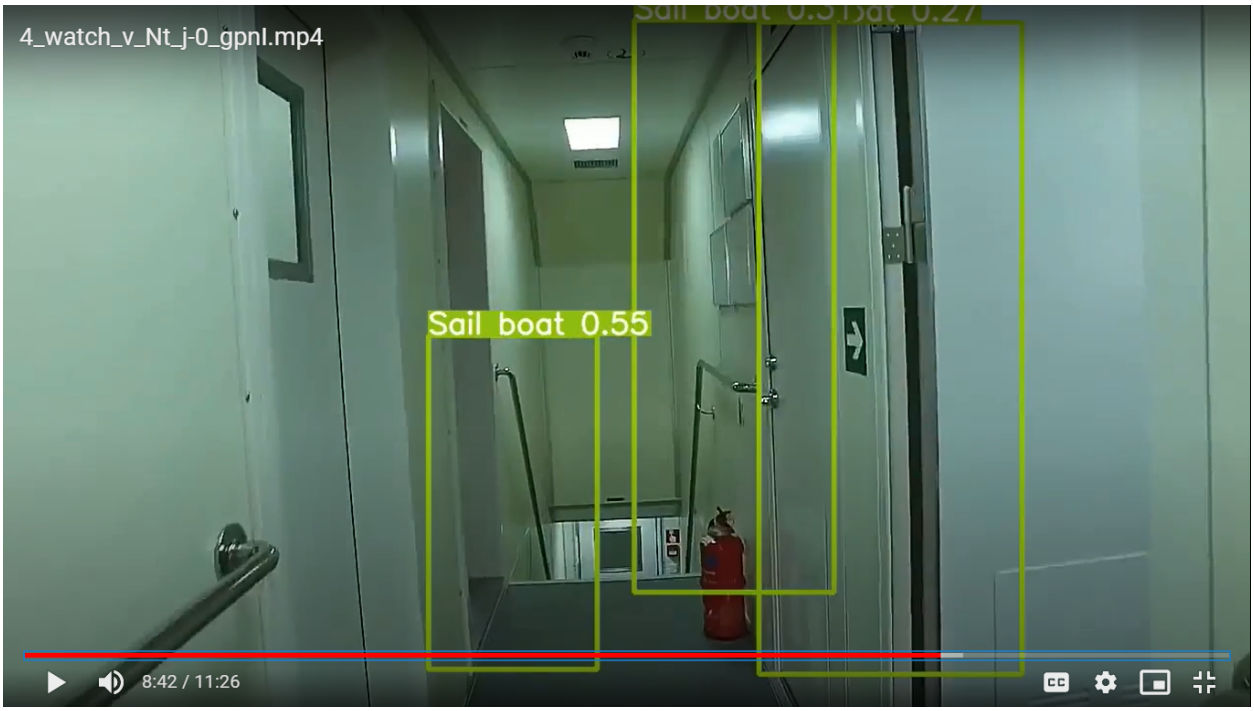


Detect Image 17. False positive: Mistakes the vessel shapes structure on top of the ship.

Detect Image 18: Correctly identifies the speed boat but fails to recognise the approaching ship (near-vertical orientation).



Detect Image 19: Incorrect Classification of birds



Detect Image 20: False positives in a different environment (other than the sea)

CONCLUSION

At the outset of the project, we built a strong foundation in deep learning and computer vision by completing the first four courses of the Deep Learning Specialization by deeplearning.ai and the "Python for Computer Vision with OpenCV and Deep Learning" course by Jose Portilla. With this grounding, we moved into the practical phase under the mentorship of Mr. Shrikant Tirki, who provided us with a dataset of 6,350 high-resolution images. These images, annotated with bounding boxes and categorized into nine maritime classes, were split into training and testing sets in a 70:30 ratio.

We first trained a YOLOv5s model using default hyperparameters and COCO-pretrained weights. Due to GColab GPU session limits, training was conducted in short bursts, culminating in 120 cumulative epochs at 640px resolution. This baseline model achieved decent performance, with a mAP@0.5 of 0.817, precision of 0.947, and recall of 0.790. However, issues such as automatic learning rate resets and a suboptimal input resolution motivated a second training phase. By manually decaying the learning rate and increasing input size to 960px, we achieved improved performance—reaching mAP@0.5 of 0.862 and mAP@0.95 of 0.552 in just 100 epochs, demonstrating faster convergence and better accuracy.

We tested the final model on two YouTube videos outside the training distribution. It performed reliably when the perspective matched the dataset—typically horizontal and long-range views—but struggled with vertical angles and close-ups, often misclassifying tall, non-ship structures as sailboats. These issues revealed limitations in the model's ability to generalize, mainly due to class imbalance and the lack of viewpoint diversity in the training set.

Overall, the project demonstrated that even within constrained GPU environments, careful tuning of hyperparameters and image resolution can significantly boost model performance. While the model shows promise for real-time maritime object detection, its generalization ability remains limited by the nature of the dataset. Addressing class imbalance and increasing viewpoint variation in future data will be key to improving robustness in real-world scenarios.

BIBLIOGRAPHY

References to the various sources utilized for this report are as follows:

1. Deep Learning Specialization, Course 1 to 4, Coursera: Andrew Ng's lecture slides, video lectures and unsolved programming exercises.
2. You Only Look Once: Unified, Real-Time Object Detection; Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi; arXiv:1506.02640 [cs.CV]
3. YOLO9000: Better, Faster, Stronger; Joseph Redmon, Ali Farhadi; arXiv:1612.08242v1 [cs.CV]
4. YOLOv3: An Incremental Improvement; Joseph Redmon, Ali Farhadi; arXiv:1804.02767 [cs.CV]
5. YOLOv4: Optimal Speed and Accuracy of Object Detection; Alexey Bochkovskiy, Chien-Yao Wang, Hong-Yuan Mark Liao; arXiv:2004.10934 [cs.CV]
6. CSPNet: A new backbone that can enhance learning capability of CNN; Wang, Chien-Yao and Mark Liao, Hong-Yuan and Wu, Yueh-Hua and Chen, Ping-Yang and Hsieh, Jun-Wei and Yeh, I-Hau
7. YOLOv5 Github repository
8. Blog article: YOLOv5 New Version - Improvements And Evaluation
9. Blog article: Data Augmentation in YOLOv4