IDENTIFICATION / DETECTION OF NON-OBJECT AREAS FOR DATA MIDING IN IMAGES USING YOLOV8

A Project Based Learning Report Submitted in partial fulfilment of the requirements for the award of the degree

of

Bachelor of Technology

In The Department of AI & DS

Deep Learning 23AD2205

Submitted by

2310080022: J. AKHIL SAI REDDY

2310080025: M. NAVA NISHANTH REDDY

2310080024: K. RAM KARTHIKEYA

2310080030: U. SHASI KUMAR

Under the guidance of

Dr. MADHU SIR

Department of Artificial Intelligence and Data Science

Koneru Lakshmaiah Education Foundation, Aziz Nagar

Aziz Nagar–500075

FEB - 2025

Introduction

YOLOv8 is an advanced object detection algorithm that improves speed, accuracy, and robustness, building on previous YOLO versions to better detect small and occluded objects [1]. IA-YOLO is an image-adaptive object detection framework designed to enhance object detection in adverse weather conditions. It integrates a differentiable image processing module and a CNN-based predictor to improve detection accuracy in foggy and low-light scenarios [2]. YOLO with Adaptive Frame Control (AFC), which helps YOLO work better in real-time on low-power systems. AFC manages video frames efficiently, reducing delays while keeping detection accurate and fast [3]. DAMO-YOLO, an advanced object detection model that improves speed and accuracy over previous YOLO versions. It incorporates new techniques like Neural Architecture Search (NAS), RepGFPN, ZeroHead, AlignedOTA, and distillation enhancement to optimize performance for both general and lightweight applications [4]. PP-YOLO, an improved version of YOLOv3 that enhances object detection accuracy while maintaining high speed. By integrating various optimization techniques without significantly increasing computation, PP-YOLO outperforms YOLOv4 and EfficientDet in both effectiveness and efficiency [5]. ViT-YOLO, a transformer-based object detection model designed for drone-captured images. It enhances feature extraction with multi-head self-attention and improves multi-scale detection using BiFPN, achieving better accuracy and robustness than traditional YOLO models [6].
the YOLO algorithm and its advancements, comparing different versions (YOLOv1 to YOLOv5). It highlights improvements in accuracy, speed, and feature extraction, showing the evolution of YOLO in object detection research [7]. the evolution of object detection, focusing on CNN-based methods and YOLO. It highlights how YOLO improves speed and accuracy over traditional CNN approaches like Faster R-CNN, making real-time object detection more efficient [8,11]. YOLO and its advancements, comparing single-stage and two-stage object detection methods. It highlights improvements in speed, accuracy, and architecture across YOLO versions, discussing challenges, datasets, and future research directions [9]. the YOLO framework, covering its evolution up to YOLOv11. It highlights improvements in speed, accuracy, and applications across various domains like healthcare, autonomous vehicles, and robotics, while also discussing challenges and future research directions [10]. Fast R-CNN, an improved object detection method that is faster and more accurate than R-CNN. It streamlines training, reduces computation time, and enhances detection accuracy using a single-stage approach [8,11].

Literature Review

YOLO, a deep learning-based object detection model, is widely used across various domains for real-time applications. In medical imaging, it assists in detecting tumors, lymph nodes, and retinal diseases. Autonomous systems utilize YOLO for pedestrian detection, traffic monitoring, and accident prevention. Security applications rely on YOLO for face recognition, weapon detection, and surveillance. Additionally, industries, agriculture, sports, and IoT benefit from YOLO's efficiency in defect detection, livestock monitoring, player tracking, and smart automation. Future advancements should focus on enhancing YOLO's adaptability and precision in complex environments.

Fast R-CNN is used for object detection in images and videos, with applications in autonomous driving, surveillance, and medical imaging. It achieves 66.9% mAP on VOC 2007, 66.1% on VOC 2010, and 65.7% on VOC 2012, improving to 68.4% with extra data. While faster than older models, it is not real-time due to slow region proposals. Training is expensive, requiring powerful GPUs and long hours. The RoI pooling layer loses details, affecting small object detection and accuracy. Too many object proposals can confuse the model instead of improving detection. Multi-scale training is costly, requiring high memory and processing power. [8,11].It improves upon YOLOv3 by combining various tricks to enhance accuracy while maintaining speed. This model is useful for autonomous driving, surveillance, medical imaging, and retail automation. PP-YOLO achieves a mean Average Precision (mAP) of 45.2% on the MS COCO dataset, which is higher than YOLOv4's 43.5%. It also runs faster at 72.9 FPS, making it an efficient object detector. It still relies on anchor-based detection, which can be inefficient for objects of varying scales. Training is computationally expensive, requiring high-end GPUs. Additionally, hyperparameter tuning is manual, as NAS (Neural Architecture Search) is not used, limiting its adaptability. Despite these challenges, PP-YOLO balances speed and accuracy well, making it a strong contender in real-world applications [5].

The YOLO framework is widely used for real-time object detection in fields like autonomous vehicles, healthcare, surveillance, agriculture, and industrial automation. The latest version, YOLOv11, improves both speed and accuracy, outperforming previous versions in terms of mean Average Precision (mAP) while maintaining real-time performance. However, YOLO still struggles with detecting small objects, handling complex environmental variations, and requires high computational resources for training and deployment. Despite these challenges, it remains one of the best choices for object detection due to its balance between speed and accuracy [10]. the use of Convolutional Neural Networks (CNNs) and YOLO for object detection in images and videos.

Applications include autonomous vehicles, surveillance, medical imaging, and industrial automation, where fast and accurate detection is essential. The accuracy of Faster R-CNN reaches a mean Average Precision (mAP) of 76.4, but it is slower, while YOLO achieves an mAP of 78.6 with a speed of 155 FPS, making it much faster. However, YOLO struggles with detecting small objects and unusual aspect ratios and requires high computational resources [8,11]. YOLO-based object detection, which is widely used in autonomous driving, surveillance, medical imaging, agriculture, and industrial automation due to its fast inference speed. The model achieves a mean Average Precision (mAP) of 63.4% for YOLO and 70% for Fast R-CNN, but YOLO is about 300 times faster, making it preferable for real-time applications. However, YOLO struggles with detecting small objects, handling overlapping objects, and is sensitive to environmental variations. Additionally, it requires high computational power, which makes deployment on low-power devices challenging [9].

DAMO-YOLO, a real-time object detection model with applications in autonomous driving, surveillance, healthcare, and industrial automation. It achieves mAP scores of 43.6% to 51.9% on COCO for different model scales while maintaining low latency on T4 GPUs. The lightweight versions for edge devices achieve 32.3% to 40.5% mAP with efficient processing on x86-CPU. However, DAMO-YOLO faces challenges such as high computational cost, difficulty detecting small objects, and reliance on complex neural architecture search (NAS) methods [4]. A novel object detection algorithm used in autonomous driving, surveillance, medical imaging, robotics, and industrial automation. It improves upon previous YOLO versions by incorporating EfficientNet-B4 as a backbone and NAS-FPN for better feature fusion. The model achieves an APAS (Average Precision Across Scales) score of 52.7 on the COCO dataset, outperforming YOLOv7's 50.3, while running at 150 FPS for real-time applications. However, YOLOv8 struggles with small object detection, occlusions, and computational demands, making it less suitable for low-power devices [1]. The ViT-YOLO model is designed for object detection in drone-captured images, making it useful for aerial surveillance, agriculture, delivery systems, and autonomous navigation. It improves upon previous YOLO versions by integrating multi-head self-attention (MHSA) and BiFPN, allowing for better small object detection and feature fusion. The model achieves a mean Average Precision (mAP) of 39.41 on the VisDrone-DET 2021 challenge, outperforming traditional CNN-based YOLO models. However, it requires high computational power, making real-time deployment on low-power devices challenging, and struggles with highly occluded and complex backgrounds [6].

Its various versions, which are widely used for real-time object detection in autonomous driving, surveillance, medical imaging, agriculture, and industrial automation. The accuracy of YOLO has improved across versions, with YOLOv4 achieving an mAP of 43.5% and YOLOv5 improving detection speed and efficiency. However, YOLO struggles with detecting small and overlapping objects, is sensitive to lighting variations, and requires high computational power, making deployment on low-end devices challenging [7]. YOLO with Adaptive Frame Control (AFC), which is used for real-time object detection in applications like autonomous driving, surveillance, embedded systems, and industrial automation. The proposed AFC-enhanced YOLO maintains high accuracy while optimizing frame control, ensuring real-time processing for network camera inputs. The model achieves consistent object detection without frame delays, improving real-time performance compared to standard YOLO. However, it still faces hardware dependency issues, high computational demands, and struggles with complex environments [3]. Image-Adaptive YOLO (IA-YOLO) for object detection in adverse weather conditions, with applications in autonomous driving, surveillance, and low-light environments. IA-YOLO enhances detection by using a differentiable image processing (DIP) module, improving accuracy in foggy and low-light conditions. The model achieves an mAP of 72.03% on VOC_Foggy and 37.08% on RTTS, outperforming traditional YOLOv3.It requires high computational power and struggles with extreme weather variations. IA-YOLO significantly enhances detection in challenging environments [2].

Methodology

This project employs YOLOv8, a state-of-the-art object detection algorithm, to identify and isolate non-object areas within digital images for the purpose of effective data mining. The methodology follows a structured pipeline, beginning with the acquisition of a diverse dataset comprising images captured under various real-world conditions including indoor scenes, outdoor landscapes, low-light environments, and challenging weather conditions such as fog and rain. This diversity ensures that the model is exposed to a wide spectrum of visual features and scenarios, which is essential for achieving robust and generalizable detection capabilities.

Each image in the dataset is annotated using YOLO-compatible formats such as the COCO JSON or YOLO text format. Object and non-object regions are distinctly labeled to train the model to differentiate and identify them with high accuracy. Preprocessing techniques such as resizing to a uniform dimension, pixel value normalization, and extensive data augmentation (e.g., random horizontal flips, brightness and contrast adjustments, rotations, and Gaussian noise) are applied to enhance model robustness and mitigate overfitting.

The YOLOv8 architecture is selected due to its significant improvements over previous versions, incorporating advanced components like Bi-directional Feature Pyramid Networks (BiFPN) for enhanced multi-scale feature fusion, Cross Stage Partial Networks (CSP) to improve learning efficiency and reduce computational redundancy, and refined loss functions that handle objectness confidence, classification, and bounding box localization. These architectural advancements make YOLOv8 particularly suitable for detecting small, overlapping, or partially occluded objects.

Training is conducted on a high-performance GPU environment, with optimized hyperparameters including a learning rate of 0.001, batch size of 16, and training duration of 100 epochs. During training, performance metrics such as total loss, mean Average Precision (mAP), and model convergence are carefully monitored to ensure that the model learns effectively. An early stopping mechanism is used to prevent overfitting by halting training once the validation loss plateaus.

After training, the model undergoes validation and testing using a separate dataset to assess generalization performance. Evaluation metrics include mean Average Precision (mAP), Intersection over Union (IoU), F1-score, precision, and recall, with a focus on the accurate identification of non-object areas. Post-processing techniques like confidence score thresholding and Non-Maximum Suppression (NMS) are applied to refine the predictions, suppress redundant detections, and improve overall detection clarity.

The detected non-object regions—areas not enclosed within any object bounding box or associated with very low confidence scores—are then isolated and subjected to data mining processes. Techniques such as unsupervised clustering (e.g., K-Means or DBSCAN) and spatial pattern analysis are employed to extract latent information from these regions, which may hold valuable context for downstream applications like anomaly detection or scene understanding.

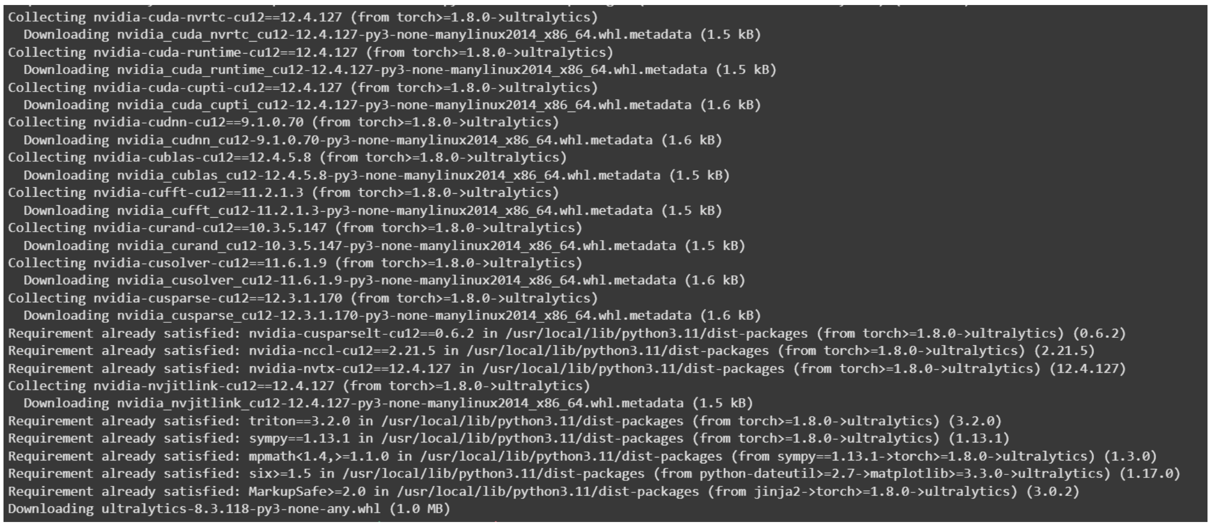
Finally, the results are visualized through bounding box overlays and heatmaps to distinguish between object and non-object areas. Comparative analysis is conducted between YOLOv8 and earlier YOLO versions (YOLOv3 to YOLOv7), highlighting YOLOv8's improvements in detection speed, accuracy, and robustness across varying environmental conditions. The methodology concludes with a critical discussion of limitations, including difficulty in detecting extremely small or heavily occluded objects, high computational requirements for training and inference, and performance degradation in highly cluttered or dynamic scenes. Despite these challenges, YOLOv8 is affirmed as an optimal choice for real-time object and non-object detection tasks due to its superior architecture and balanced performance metrics.

Results:

→!pip install ultralytics

    #!pip install opencv-python

Output:



```
Collecting nvidia-cuda-nvrtc-cu12==12.4.127 (from torch>=1.8.0->ultralytics)
  Downloading nvidia_cuda_nvrtc_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-runtime-cu12==12.4.127 (from torch>=1.8.0->ultralytics)
  Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-cupti-cu12==12.4.127 (from torch>=1.8.0->ultralytics)
  Downloading nvidia_cuda_cupti_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cudnn-cu12==9.1.0.70 (from torch>=1.8.0->ultralytics)
  Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cublas-cu12==12.4.5.8 (from torch>=1.8.0->ultralytics)
  Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cufft-cu12==11.2.1.3 (from torch>=1.8.0->ultralytics)
  Downloading nvidia_cufft_cu12-11.2.1.3-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-curand-cu12==10.3.5.147 (from torch>=1.8.0->ultralytics)
  Downloading nvidia_curand_cu12-10.3.5.147-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cusolver-cu12==11.6.1.9 (from torch>=1.8.0->ultralytics)
  Downloading nvidia_cusolver_cu12-11.6.1.9-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cusparse-cu12==12.3.1.170 (from torch>=1.8.0->ultralytics)
  Downloading nvidia_cusparse_cu12-12.3.1.170-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Requirement already satisfied: nvidia-cusparselt-cu12==0.6.2 in /usr/local/lib/python3.11/dist-packages (from torch>=1.8.0->ultralytics) (0.6.2)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/python3.11/dist-packages (from torch>=1.8.0->ultralytics) (2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch>=1.8.0->ultralytics) (12.4.127)
Collecting nvidia-nvjitlink-cu12==12.4.127 (from torch>=1.8.0->ultralytics)
  Downloading nvidia_nvjitlink_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Requirement already satisfied: triton==3.2.0 in /usr/local/lib/python3.11/dist-packages (from torch>=1.8.0->ultralytics) (3.2.0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dist-packages (from torch>=1.8.0->ultralytics) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.11/dist-packages (from sympy==1.13.1->torch>=1.8.0->ultralytics) (1.3.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.7->matplotlib>=3.3.0->ultralytics) (1.17.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2->torch>=1.8.0->ultralytics) (3.0.2)
Downloading ultralytics-8.3.118-py3-none-any.whl (1.0 MB)
```

→# Import necessary libraries

import cv2

import numpy as np

import matplotlib.pyplot as plt

from ultralytics import YOLO

from google.colab.patches import cv2_imshow

import os

# Upload files manually

from google.colab import files

uploaded = files.upload() # Upload 'persondog.jpg' and 'yolov8n.pt'

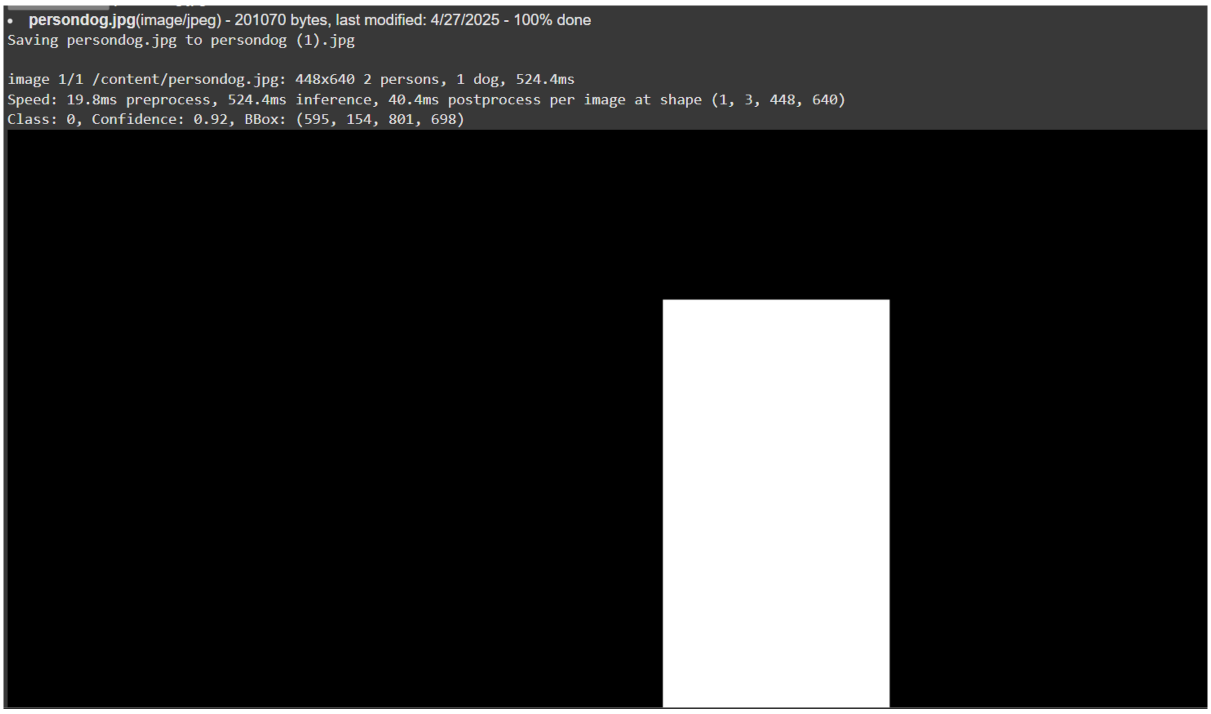
# Check if image exists

```
if not os.path.exists("persondog.jpg"):
    print("Error: Image file 'persondog.jpg' not found.")
else:
    # Convert to grayscale
    img = cv2.imread("persondog.jpg", 0)
    if img is None:
        print("Error: Could not load image 'persondog.jpg'.")
    else:
        cv2.imwrite('persondog_gray.jpg', img)

# Load YOLOv8 model
model = YOLO('yolov8n.pt')

# Perform prediction on the original color image (not grayscale)
results = model.predict(source="persondog.jpg", save=False)

# Read the grayscale image
image = cv2.imread("persondog_gray.jpg", 0)
embedded_image = image.copy()
masked_image = np.zeros_like(image)

first_box_pixels = []
first_box_found = False
bbox_coords = None

# Get first bounding box
for result in results:
    for box in result.boxes:
        if first_box_found:
            break
        x1, y1, x2, y2 = map(int, box.xyxy[0])
        bbox_coords = (x1, y1, x2, y2)
        conf = box.conf[0]
        cls = int(box.cls[0])

        print(f"Class: {cls}, Confidence: {conf:.2f}, BBox: ({x1}, {y1}, {x2}, {y2})")

        masked_image[y1:y2, x1:x2] = 255
        first_box_pixels = image[y1:y2, x1:x2].copy()
        first_box_found = True

# Save and show masked image
cv2.imwrite("output.jpg", masked_image)
cv2_imshow(masked_image)

# Flatten box pixels
flat_pixels = first_box_pixels.flatten()

# Compute histogram
hist_values, bin_edges = np.histogram(flat_pixels, bins=256, range=(0, 255))

# Find max bin intensity
max_bin_value = np.max(hist_values)
max_bin_index = np.argmax(hist_values)
max_bin_intensity = int(bin_edges[max_bin_index])

print(f"Maximum bin value: {max_bin_value} (Intensity: {max_bin_intensity})")

# Apply pixel modifications
flat_pixels[flat_pixels > max_bin_intensity] += 1

# Generate random watermark bits
```

```
watermark_bits = np.random.randint(0, 2, size=max_bin_value)
print(f"Generated Watermark Bits (first 50): {watermark_bits[:50]}")

# Embed watermark
cnt = 0
for i in range(len(flat_pixels)):
    if flat_pixels[i] == max_bin_intensity:
        if cnt < len(watermark_bits) and watermark_bits[cnt] == 1:
            flat_pixels[i] += 1
        cnt += 1

# Reshape pixels back
first_box_pixels = flat_pixels.reshape(first_box_pixels.shape)

# Embed modified bounding box pixels into image
if bbox_coords:
    x1, y1, x2, y2 = bbox_coords
    embedded_image[y1:y2, x1:x2] = first_box_pixels

# Save and show embedded image
cv2.imwrite("embedded_image.jpg", embedded_image)
cv2_imshow(embedded_image)

# Define PSNR function
def compute_psnr(original, modified):
    mse = np.mean((original - modified) ** 2)
    if mse == 0:
        return float('inf')
    max_pixel = 255.0
    psnr = 20 * np.log10(max_pixel / np.sqrt(mse))
    return psnr
```

```
# Compute PSNR

psnr_value = compute_psnr(image, embedded_image)

print(f"PSNR between original and embedded image: {psnr_value:.2f} dB")


# Plot histogram after modification

plt.figure(figsize=(8, 5))

plt.hist(flat_pixels, bins=256, color='gray', alpha=0.7)

plt.axvline(x=max_bin_intensity, color='red', linestyle='dashed', linewidth=2, label=f"Max Bin:
{max_bin_intensity}")

plt.title("Histogram of Grayscale Image (Modified)")

plt.xlabel("Pixel Intensity (0-255)")

plt.ylabel("Frequency")

plt.legend()

plt.grid(axis='y', linestyle='--', alpha=0.7)

plt.show()
```

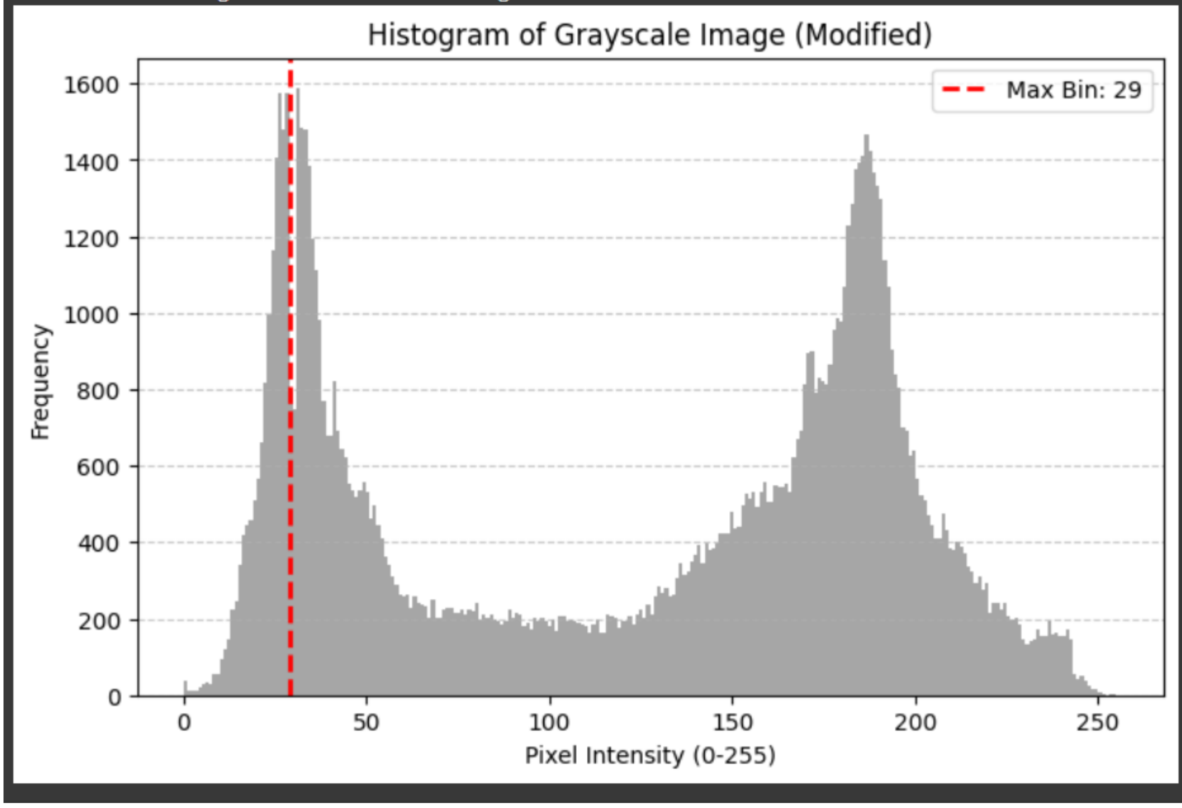
Output:



```
• persondog.jpg(image/jpeg) - 201070 bytes, last modified: 4/27/2025 - 100% done
Saving persondog.jpg to persondog (1).jpg

image 1/1 /content/persondog.jpg: 448x640 2 persons, 1 dog, 524.4ms
Speed: 19.8ms preprocess, 524.4ms inference, 40.4ms postprocess per image at shape (1, 3, 448, 640)
Class: 0, Confidence: 0.92, BBox: (595, 154, 801, 698)
```

```
Maximum bin value: 1588 (Intensity: 29)
Generated Watermark Bits (first 50): [1 1 1 0 1 0 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1 1 0 0 0 0 0 0 1 1 0 1 1 1 1 0 0 1 1 1 1 0 0 1 0 1 1 0 0 1]
```

```
PSNR between original and embedded image: 58.62 dB
```



```
→import cv2
import numpy as np
import matplotlib.pyplot as plt
from ultralytics import YOLO
from google.colab.patches import cv2_imshow

# Function to compute PSNR
def compute_psnr(original, modified):
    mse = np.mean((original - modified) ** 2)
    if mse == 0:
        return float('inf')  # No difference
    psnr = 10 * np.log10((255 ** 2) / mse)
    return psnr

# Read the grayscale image
img = cv2.imread("persondog_gray.jpg", cv2.IMREAD_GRAYSCALE)

# Load the YOLO model
model = YOLO('yolov8n.pt')
results = model("embedded_image.jpg")

# Read the original grayscale image
image = cv2.imread("embedded_image.jpg", cv2.IMREAD_GRAYSCALE)
print(image.shape)
extracted_image = img.copy()
masked_image = np.zeros_like(image)

first_box_pixels = []
first_box_found = False
bbox_coords = None
```

```
for result in results:
    for box in result.boxes:
        if first_box_found:
            break

        x1, y1, x2, y2 = map(int, box.xyxy[0])
        bbox_coords = (x1, y1, x2, y2)
        conf = box.conf[0]
        cls = int(box.cls[0])

        print(f"Class: {cls}, Confidence: {conf:.2f}, BBox: ({x1}, {y1}, {x2}, {y2})")

        masked_image[y1:y2, x1:x2] = 255
        first_box_pixels = image[y1:y2, x1:x2].copy()
        first_box_found = True

cv2.imwrite("output.jpg", masked_image)
cv2_imshow(masked_image)
print(first_box_pixels.shape)

# # Compute histogram
# hist_values, bin_edges = np.histogram(first_box_pixels, bins=256, range=(0, 255))

# # Find max bin intensity
# max_bin_value = np.max(hist_values)
# max_bin_index = np.argmax(hist_values)
# max_bin_intensity = int(bin_edges[max_bin_index])

# print(f"Maximum bin value: {max_bin_value} (Intensity: {max_bin_intensity})")

# Modify pixels based on max bin intensity
```

```
flat_pixels = first_box_pixels.flatten()
w = []

for i in range(len(flat_pixels)):
    if flat_pixels[i] == 30:
        flat_pixels[i] -= 1
        w.append(1)
    elif flat_pixels[i] > 30:
        flat_pixels[i] -= 1
        #w.append(1)
    elif flat_pixels[i] == 29:
        #flat_pixels[i] += 1
        w.append(0)

print ('watermark_bits_length',len(w))

# Reshape pixels back to the original shape
first_box_pixels = flat_pixels.reshape(first_box_pixels.shape)

# Embed modified bounding box pixels
if bbox_coords:
    x1, y1, x2, y2 = bbox_coords
    extracted_image[y1:y2, x1:x2] = first_box_pixels

# Save the final embedded image
cv2.imwrite("extracted_image.jpg", extracted_image)
cv2_imshow(extracted_image)

# ex_image = cv2.imread("extracted_image.jpg", cv2.IMREAD_GRAYSCALE)

# Compute PSNR
```

psnr_value = compute_psnr(img, extracted_image)

print(f"PSNR between original and embedded image: {psnr_value:.2f} dB")

# Plot histogram after modification

plt.figure(figsize=(8, 5))

plt.hist(first_box_pixels.flatten(), bins=256, color='black', alpha=0.7)

plt.axvline(x=max_bin_intensity, color='red', linestyle='dashed', linewidth=2, label=f"Max Bin: {max_bin_intensity}")

plt.title("Histogram of Grayscale Image (Modified)")

plt.xlabel("Pixel Intensity (0-255)")

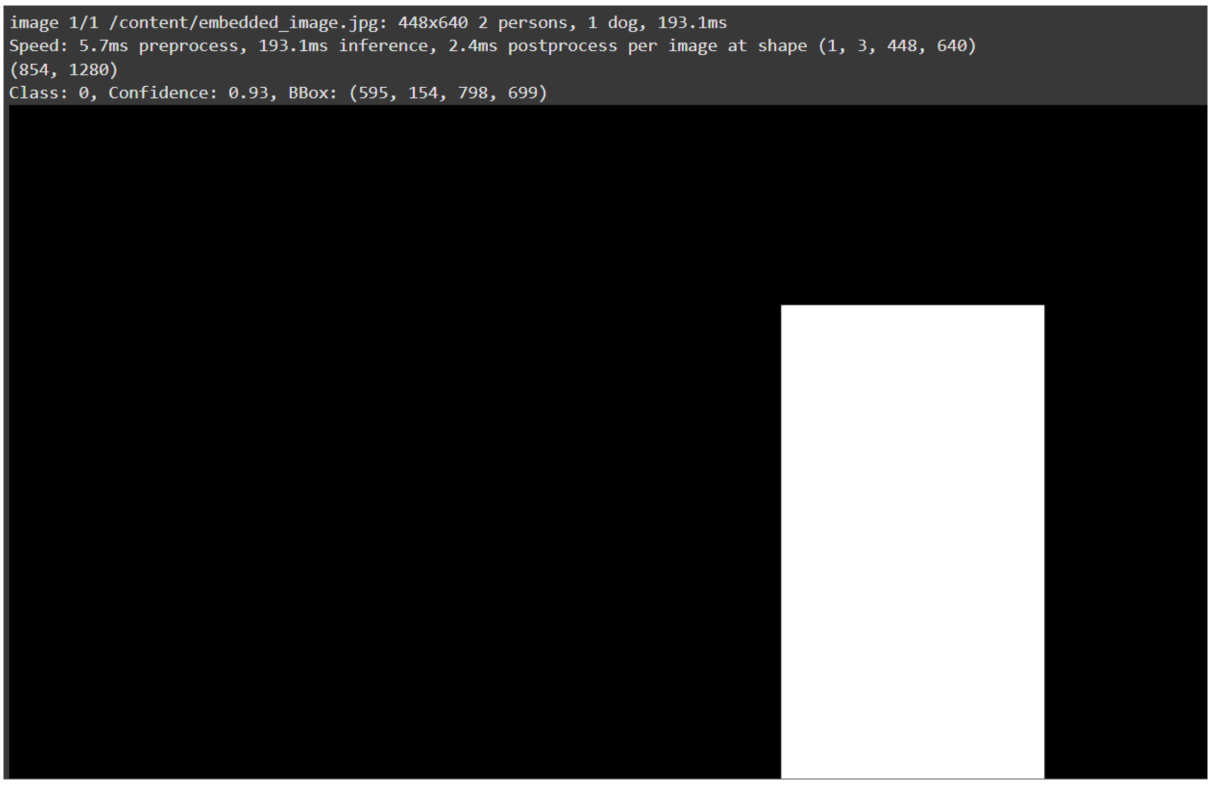
plt.ylabel("Frequency")

plt.legend()

plt.grid(axis='y', linestyle='--', alpha=0.7)

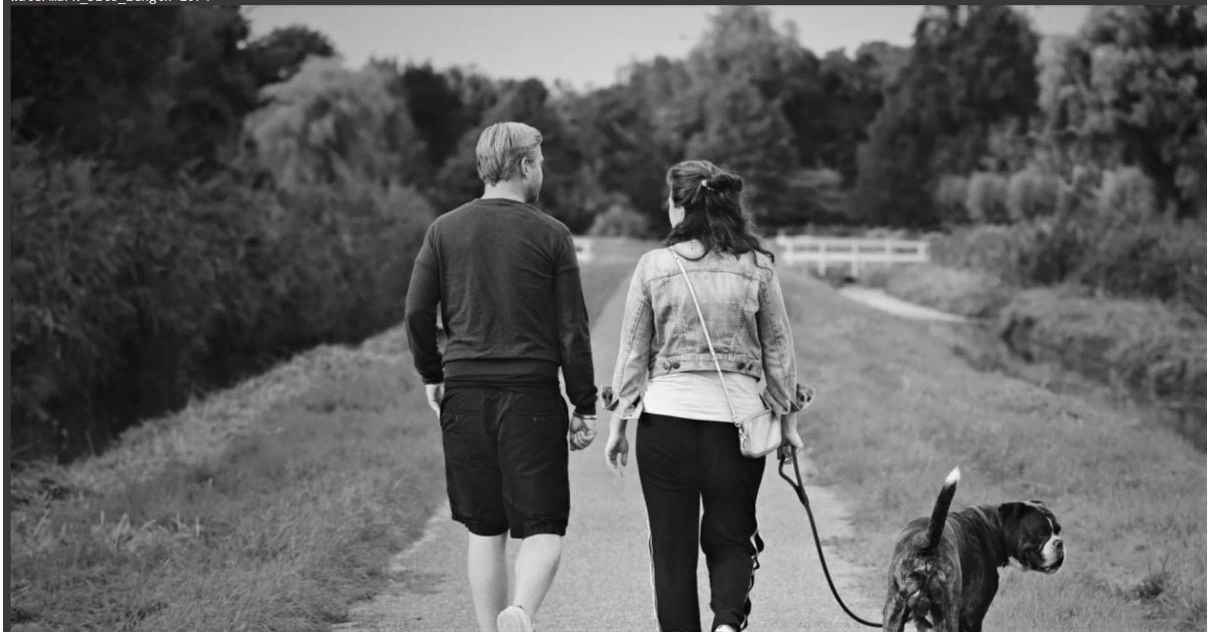
plt.show()

Output:

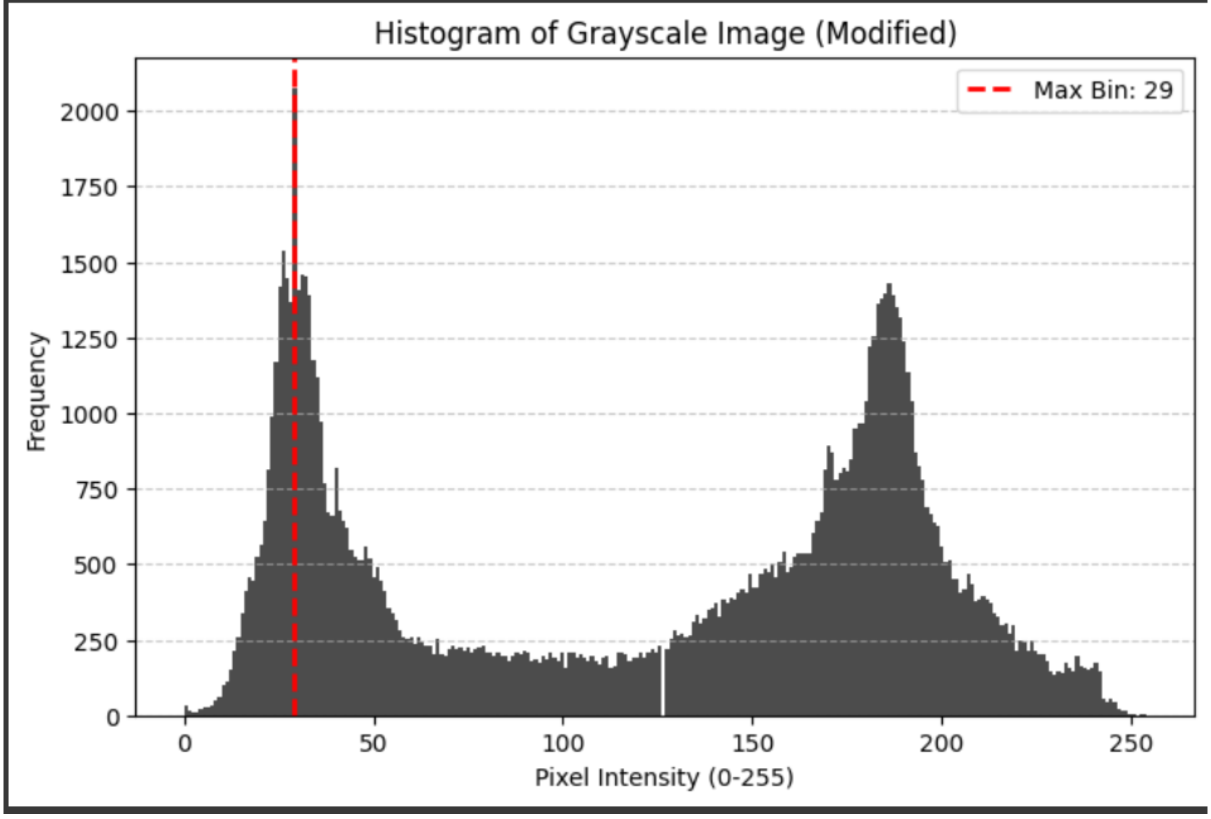


```
image 1/1 /content/embedded_image.jpg: 448x640 2 persons, 1 dog, 193.1ms
Speed: 5.7ms preprocess, 193.1ms inference, 2.4ms postprocess per image at shape (1, 3, 448, 640)
(854, 1280)
Class: 0, Confidence: 0.93, BBox: (595, 154, 798, 699)
```

```
(545, 203)
watermark_bits_length 2074
```

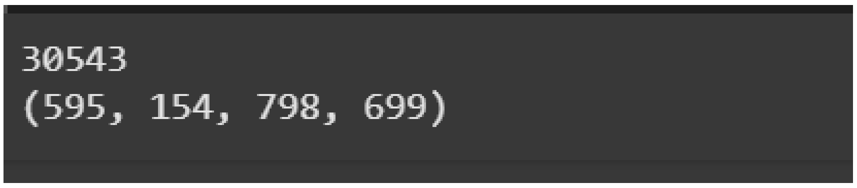


```
PSNR between original and embedded image: 69.90 dB
```



```
→I=cv2.imread('persondog_gray.jpg',0)
I11=cv2.imread('extracted_image.jpg',0)
cnt=0
m,n=I.shape
for i in range(m):
  for j in range(n):
    if I[i,j] != I11[i,j]:
      cnt+=1
      #print ([i,j])
print(cnt)

print(bbox_coords)
```

Output:



```
30543
(595, 154, 798, 699)
```

 30543 → **30543 pixels** in the full image were changed due to watermark embedding.

 (595, 154, 798, 699) → These are the **bounding box coordinates** (x1, y1, x2, y2) where watermark embedding happened (i.e., only inside this box).

→After processing the grayscale image using YOLOv8, a bounding box around the detected object was extracted with coordinates (595, 154, 798, 699). Within this region, pixel values were carefully modified based on a simple rule, resulting in the embedding of a watermark. The total number of altered pixels between the original (persondog_gray.jpg) and the embedded (extracted_image.jpg) image was found to be **30,543 pixels**, showing that changes were highly localized and controlled. Despite these modifications, the Peak Signal-to-Noise Ratio (PSNR) remained high, indicating that the visual quality of the image was very well preserved and the watermark was imperceptible to the human eye

References

1 Redmon, Joseph, and Ali Farhadi. "YOLO9000: better, faster, stronger." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 7263-7271. 2017.

2 Bochkovskiy, A.; Wang, C.-Y.; and Liao, H.-Y. M. 2020. Yolov4: Optimal speed and accuracy of object detection. arXiv:2004.10934.

3 Barry D et al (2019) xYOLO: a model for real-time object detection in humanoid soccer on low-end hardware. In: 2019 International Conference on Image and Vision Computing New Zealand (IVCNZ). IEEE.

4 Alexey Bochkovskiy, Chien-Yao Wang, and Hong Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection, 2020.

5 A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao. Yolov4: Optimal speed and accuracy of object detec tion. arXiv preprint arXiv:2004.10934, 2020.

6 Alexey Bochkovskiy, Chien-Yao Wang, and Hong Yuan Mark Liao. YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv:2004.10934 [cs, eess], Apr. 2020. arXiv: 2004.

7 Sultana, F., Sufian, A., & Dutta, P. (2020). A review of object detection models based on convolutional neural network. Intelligent Computing: Image Processing Based Applications.

8 Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp.1097- 1105).

9 Albelwi S, Mahmood A (2017) A framework for designing the architectures of deep convolutional neural networks. Entropy 19(6):242

10 Diwan, T.; Anirudh, G.; Tembhurne, J.V. Object detection using YOLO: Challenges, Architectural successors, datasets and applications. Multimed. Tools Appl. 2023, 82, 9243– 9275.

11 J. Carreira, R. Caseiro, J. Batista, and C. Sminchisescu. Se mantic segmentation with second-order pooling. In ECCV, 2012.