**Experimental Setup**

The experimental setup for this research consists of a live drone-based aerial monitoring system integrated with a ground station for real-time image processing and adaptive traffic signal control. This setup uses RTMP live streaming to relay video feeds from a drone to a processing unit on the ground, where a sequence of image and data processing steps are applied to analyze vehicle types, densities, and traffic conditions. This information is then used to prioritize and schedule signal releases for optimizing traffic flow. The following sections describe each aspect of the setup in detail.

**1. Drone and Ground Station Configuration**

To acquire real-time traffic data, we use a DJI drone to capture aerial footage of the traffic intersections under study. The drone streams video to the ground station using RTMP (Real-Time Messaging Protocol), allowing for seamless transfer of live images with minimal delay. The ground station serves as the computational hub, where the incoming video feed is processed to identify vehicles, analyze traffic density, and manage signal scheduling based on adaptive traffic flow algorithms.

The MonaServer setup on the ground station allows the drone’s video feed to be processed directly. To access this stream, the OpenCV library is used for reading the RTMP feed, while the YOLOv8 model is applied to detect and annotate vehicles in real-time.

The following Python script initiates the RTMP feed, performs vehicle detection using YOLOv8, and displays annotated frames showing detected objects with bounding boxes and labels.

import cv2

from ultralytics import YOLO

rtmp\_url = 'rtmp://192.168.209.12:1935/live'

cap = cv2.VideoCapture(rtmp\_url)

model = YOLO('yolo11n.pt')

if not cap.isOpened():

    print("Error: Could not open video stream.")

    exit()

while cap.isOpened():

    ret, frame = cap.read()

    if ret:

        results = model(frame)

        annotated\_frame = results.render()[0]

        cv2.imshow('RTMP Stream with YOLOv8', annotated\_frame)

        if cv2.waitKey(1) & 0xFF == ord('q'):

            break

    else:

        break

cap.release()

cv2.destroyAllWindows()

**2. Region of Interest (ROI) Selection for Signal Detection**

The first step in processing the drone feed is to define Regions of Interest (ROIs) around each traffic signal in the monitored area. Each ROI corresponds to a specific signal location and serves as the area where vehicle detection and counting will occur.

A custom Python script, developed with OpenCV, enables manual selection and saving of these ROIs as coordinates. Each ROI is defined by drawing a rectangular bounding box over the target area in the image. The coordinates of each bounding box are then saved in a CSV file, along with details of the image, ROI ID, and dimensions. This CSV file is later referenced during vehicle detection to confine the object detection process to the selected ROIs.

**Code for ROI Selection:**

By isolating specific areas around each signal, the model avoids analyzing irrelevant sections of the image, making the process more efficient and reducing the possibility of false detections outside the traffic signal areas.

import cv2

import numpy as np

import os

import csv

def select\_rois(image\_path):

    # Read the image

    image = cv2.imread(image\_path)

    if image is None:

        print(f"Error: Unable to read image at {image\_path}")

        return None

    # Create a window

    window\_name = f"Select ROIs - {os.path.basename(image\_path)}"

    cv2.namedWindow(window\_name)

    # Create a copy of the image for drawing

    img\_copy = image.copy()

    overlay = image.copy()

    # Variables to store rectangle properties

    rect\_start = None

    rect\_end = None

    drawing = False

    rois = []

    def mouse\_callback(event, x, y, flags, param):

        nonlocal rect\_start, rect\_end, drawing, img\_copy, overlay

        if event == cv2.EVENT\_LBUTTONDOWN:

            rect\_start = (x, y)

            drawing = True

            overlay = img\_copy.copy()

        elif event == cv2.EVENT\_MOUSEMOVE:

            if drawing:

                overlay = img\_copy.copy()

                cv2.rectangle(overlay, rect\_start, (x, y), (0, 255, 0), 2)

        elif event == cv2.EVENT\_LBUTTONUP:

            rect\_end = (x, y)

            drawing = False

            cv2.rectangle(img\_copy, rect\_start, rect\_end, (0, 255, 0), 2)

            overlay = img\_copy.copy()

    cv2.setMouseCallback(window\_name, mouse\_callback)

    print("\nInstructions:")

    print("- Click and drag to draw an ROI")

    print("- Press 'a' to add the current ROI")

    print("- Press 'r' to reset all ROIs")

    print("- Press 's' to save all ROIs and move to the next image")

    print("- Press 'q' to quit without saving")

    while True:

        display\_img = overlay.copy()

        cv2.imshow(window\_name, display\_img)

        key = cv2.waitKey(1) & 0xFF

        if key == ord('q'):

            cv2.destroyAllWindows()

            return None

        elif key == ord('a') and rect\_start and rect\_end:

            roi = (min(rect\_start[0], rect\_end[0]),

                   min(rect\_start[1], rect\_end[1]),

                   abs(rect\_end[0] - rect\_start[0]),

                   abs(rect\_end[1] - rect\_start[1]))

            rois.append(roi)

            print(f"ROI added: x={roi[0]}, y={roi[1]}, width={roi[2]}, height={roi[3]}")

            rect\_start = None

            rect\_end = None

        elif key == ord('r'):

            rois = []

            img\_copy = image.copy()

            overlay = img\_copy.copy()

            print("All ROIs reset")

        elif key == ord('s'):

            cv2.destroyAllWindows()

            return rois

def process\_images(image\_folder, output\_csv):

    # Ensure the image folder exists

    if not os.path.isdir(image\_folder):

        print(f"Error: The folder {image\_folder} does not exist.")

        return

    # Get all image files in the folder

    image\_files = [f for f in os.listdir(image\_folder) if f.lower().endswith(('.png', '.jpg', '.jpeg'))]

    if not image\_files:

        print(f"No image files found in {image\_folder}")

        return

    # Prepare CSV file

    with open(output\_csv, 'w', newline='') as csvfile:

        csvwriter = csv.writer(csvfile)

        csvwriter.writerow(['image\_name', 'roi\_id', 'roi\_x', 'roi\_y', 'roi\_width', 'roi\_height'])

        for image\_file in image\_files:

            image\_path = os.path.join(image\_folder, image\_file)

            print(f"\nProcessing: {image\_file}")

            rois = select\_rois(image\_path)

            if rois:

                for i, roi in enumerate(rois):

                    csvwriter.writerow([image\_file, i+1, roi[0], roi[1], roi[2], roi[3]])

                print(f"{len(rois)} ROIs saved for {image\_file}")

            else:

                print(f"Skipped {image\_file}")

    print(f"\nROI information saved to {output\_csv}")

if \_\_name\_\_ == "\_\_main\_\_":

    image\_folder = "./images"  # Replace with your image folder path

    output\_csv = "./roi\_info.csv"  # Name of the output CSV file

    process\_images(image\_folder, output\_csv)

**3. Vehicle Detection Using YOLOv11 Model**

Once the ROIs are set, each frame from the drone feed is processed through a trained YOLOv11 (You Only Look Once) object detection model to identify and classify vehicles within the ROIs. YOLOv11, chosen for its high accuracy and speed, is trained on a custom dataset, including vehicle types relevant to urban traffic scenarios, such as cars, buses, trucks, motorcycles, and bicycles. The model outputs bounding boxes and class labels for each detected vehicle.

The code reads each ROI from the CSV file, applies the YOLOv11 model to the specific region, and counts the occurrences of each vehicle type within that ROI. This approach confines vehicle detection to critical areas, ensuring that only vehicles relevant to each traffic signal are considered.

**Code for Vehicle Detection in ROIs:**

import csv

from ultralytics import YOLO

import cv2

# Initialize YOLO model

model = YOLO("yolo11n.pt")

# Load the image of a traffic junction

image\_path = "./images/sample.jpg"

image = cv2.imread(image\_path)

# Load ROI data from the CSV file

roi\_file = './roi\_selection/roi\_info.csv'

roi\_data = []

with open(roi\_file, mode='r') as file:

    reader = csv.DictReader(file)

    for row in reader:

        if row['image\_name'] == 'sample.jpg':  # Filter rows for the current image

            roi\_data.append([int(row['roi\_id']), int(row['roi\_x']), int(row['roi\_y']), int(row['roi\_width']), int(row['roi\_height'])])

# Perform inference on the image

results = model([image])

# Initialize dictionary to hold object counts in each ROI

object\_counts = {}

# Define object classes based on custom dataset

visdrone\_classes = ['pedestrian', 'car', 'van', 'bus', 'bike']

# Iterate through each ROI from the CSV data

for roi in roi\_data:

    roi\_id, roi\_x, roi\_y, roi\_width, roi\_height = roi

    # Create an empty dictionary to hold the object counts for this ROI

    object\_counts[f"s{roi\_id}"] = {class\_name: 0 for class\_name in visdrone\_classes}

    # Define the bounding box for the ROI

    roi\_bbox = (roi\_x, roi\_y, roi\_x + roi\_width, roi\_y + roi\_height)

    # Process each result (bounding boxes)

    for result in results:

        boxes = result.boxes  # Get detected bounding boxes

        for box in boxes:

            xyxy = box.xyxy[0].cpu().numpy()  # Bounding box coordinates (Xmin, Ymin, Xmax, Ymax)

            cls\_idx = int(box.cls[0].cpu().numpy())  # Class index

            class\_name = visdrone\_classes[cls\_idx]  # Class name

            # Check if the bounding box is inside the ROI

            xmin, ymin, xmax, ymax = xyxy

            if xmin >= roi\_bbox[0] and ymin >= roi\_bbox[1] and xmax <= roi\_bbox[2] and ymax <= roi\_bbox[3]:

                # Increment count for the detected object

                object\_counts[f"s{roi\_id}"][class\_name] += 1

# Print the resulting dictionary with object counts in each ROI

print(object\_counts)

**4. Calculation of Weighted Vehicle Counts**

Each detected vehicle is assigned a weight based on its type, which represents the area it occupies on the road and its influence on traffic flow. For instance, larger vehicles like buses and trucks have higher weights compared to smaller vehicles like motorcycles or bicycles. The weights are stored in a dictionary, with values assigned based on the typical impact of each vehicle type on intersection clearance time.

The product of each vehicle’s weight and count is computed for each ROI, generating a total “score” for each signal. This score helps determine the priority of signal release based on the cumulative impact of waiting vehicles.

**Code for Weighted Sum Calculation:**

def calculate\_weighted\_sum(input\_dict):

    vehicle\_weights = {'pedestrian': 1, 'bike': 5, 'car': 20,'bus': 100}

    result = {}

    for section, vehicles in input\_dict.items():

        weighted\_sum = sum(count \* vehicle\_weights[vehicle\_type] for vehicle\_type, count in vehicles.items())

        result[section] = weighted\_sum

    return result

**5. Scheduling Signal Release Based on Priority**

After calculating the weighted sums for each ROI, the scores are used to schedule the release of traffic signals. A priority-based scheduling algorithm ranks the signals in descending order of their weighted sums. Signals with higher scores are released first, as they indicate higher vehicle accumulation and, thus, greater urgency for traffic flow. The duration of each signal release is dynamically set based on the number of vehicles detected in the ROI.

The ground station continuously evaluates all signals, assigning priority to the signal with the highest weighted score and adjusting release times to prevent repetitive serving of the same signal. This adaptive scheduling approach enables optimal flow across multiple intersections, reducing overall wait times.

**Code for Scheduling Signal Release:**

def generate\_vehicle\_schedule(weighted\_sums):

    current\_time = datetime.now()

    schedule = {}

    sorted\_sections = sorted(weighted\_sums.items(), key=lambda x: x[1], reverse=True)

    for section, weight in sorted\_sections:

        duration = timedelta(seconds=weight \* 0.2)

        end\_time = current\_time + duration

        schedule[section] = {

            'start\_time': current\_time.strftime('%Y-%m-%d %H:%M:%S'),

            'end\_time': end\_time.strftime('%Y-%m-%d %H:%M:%S'),

            'duration': str(duration)

        }

        current\_time = end\_time

    return schedule

**Summary of Experimental Setup**

In summary, the experimental setup integrates aerial imagery, real-time data streaming, region-based vehicle detection, and adaptive signal scheduling. By marking and processing specific ROIs, applying YOLOv11 for object detection, and using a weighted scoring system, this setup achieves an effective mechanism for optimizing traffic signal operations. The described approach offers a scalable framework that can be adapted to different traffic scenarios and urban environments, providing a practical solution to alleviate congestion and improve urban mobility.

This setup forms the foundation of an adaptive traffic control system that leverages cutting-edge computer vision and machine learning technologies to deliver a data-driven response to dynamic urban traffic conditions.