**TRAFFIC MANAGEMENT SYSTEMS**

**PHASE -1**

**TRAFFIC MANAGEMENT WITH IoT :**

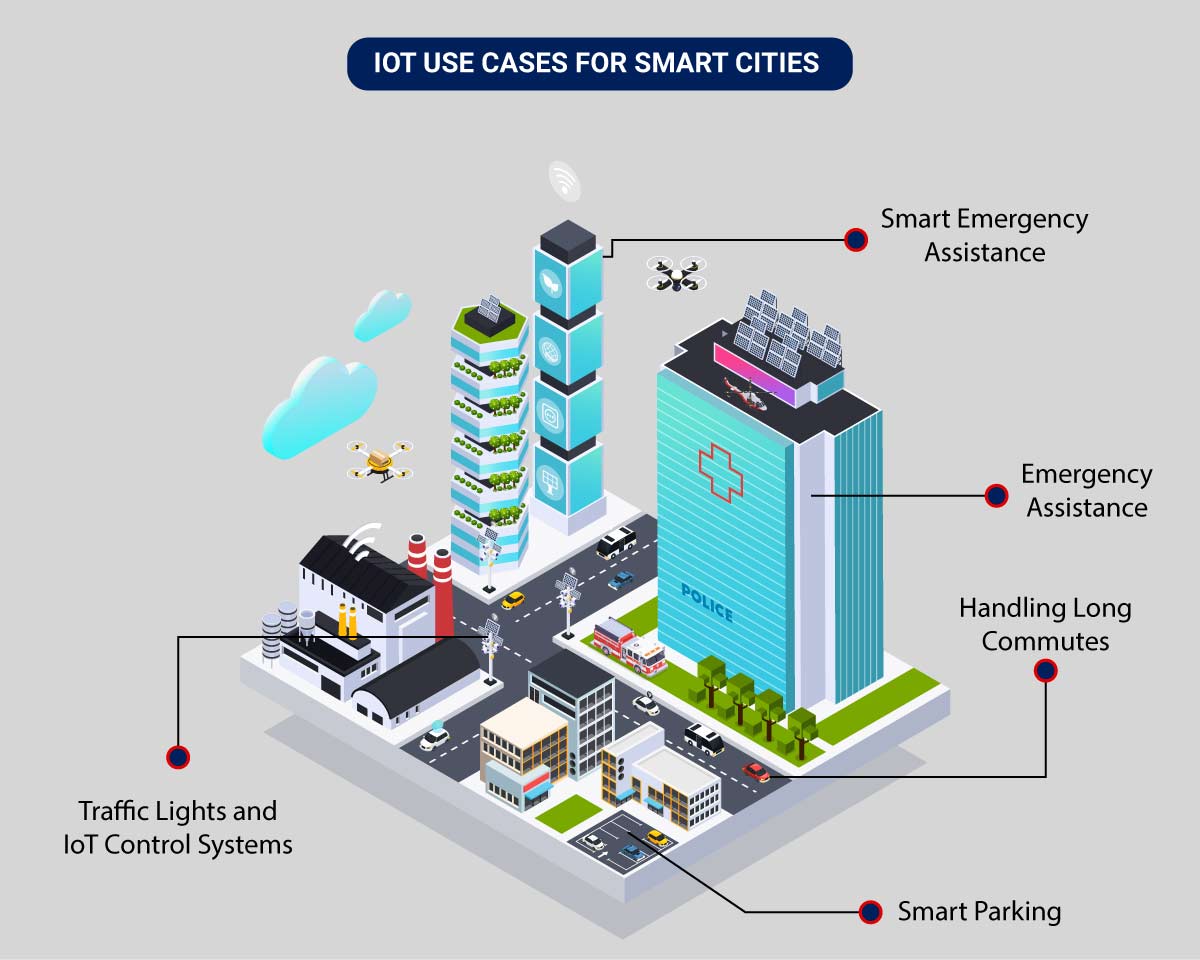
****

**Role of IoT in Smart City Traffic Management :**

* Expand the capacity of city streets without having to build new roads.
* Optimize the traffic flow and keep the drivers safe. It would include cameras, sensors, and cellular technologies that automatically adjust traffic lights, expressway lanes, speed limits, and highway exit counters.
* Transmit accurate information about available parking spaces to citizens in real-time
* Collect data on congestion and improve traffic signaling to reduce blockages and optimize commute
* Locate incidents and report them to emergency rooms immediately with road sensors and video surveillance
* Employ real-time data feeds to ensure the streetlights turn dim or brighten up per the changing weather conditions and the onset of day and night.

## Application of IoT in Traffic Management :

City governments can improve their operations & infrastructure by placing IoT sensors and tracking devices on roads and highways for recording, analyzing, and sharing data in real-time.



* **Traffic Lights and IoT Control Systems**: Smart traffic signals may look like a typical stoplight, yet they utilize an array of sensors to monitor real-time traffic. Usually, the goal is to help cars reduce the amount of time spent idle. And IoT technology enables the various signals to communicate with each other. This is while adapting to changing traffic conditions in real time. The outcome is less time spent in traffic jams and even reduced carbon emissions.
* **Parking Enabled through IoT**: Smart meters and mobile apps make on-street parking spaces easily accessible with instant notifications. Drivers receive alerts whenever a parking spot is available to reserve it instantly. The app gives easy directions to the parking spot with a convenient online payment option.
* **Emergency Assistance through IoT**: A traffic monitoring system using IoT technology enables emergency responders to speed up the care mechanism in case of accidents late at night or in isolated locations. The sensors on the road detect any accident, and the problem is immediately reported to the traffic management system. This request is passed on to relevant authorities to take corrective action. Emergency response personnel would include medical technicians, police officers, and fire departments for enhanced responsiveness and timely intervention.
* **Commute Assistance:**With every vehicle acting as an IoT sensor, a dedicated app can make suggestions, determine optimal routes & provide advance notice of accidents or traffic jams. Further, it can even suggest the best time to leave. It is all because of a robust algorithm that helps reduce driving time with intelligent traffic lights.

**IoT-based Traffic Management System**

**involves multiple components and technologies. Below, I'll outline a high-level project plan for such a system**:

**Project Overview:**

Build an IoT-based Traffic Management System to monitor, analyze, and manage traffic at intersections. The system will collect real-time data, optimize traffic flow, and provide insights for better decision-making.

**Components:**

**A. IoT Sensors and Cameras:**

Sensors and cameras are essential components of a Traffic Management System. These devices play a crucial role in collecting real-time data from traffic intersections. Here's an overview of common IoT sensors and cameras used in such systems:

1. Inductive Loop Sensors:

   - Inductive loop sensors are embedded in the road surface.

   - They detect the presence of vehicles by measuring changes in inductance.

   - Used to count vehicles, measure traffic flow, and detect congestion.

2. Ultrasonic Sensors:

   - Ultrasonic sensors use sound waves to measure the distance to objects, including vehicles.

   - They can provide data on the distance between vehicles and vehicle count.

3. Infrared Sensors:

   - Infrared sensors detect the heat emitted by vehicles.

   - Used for vehicle presence detection at intersections.

4. Lidar Sensors:

   - Lidar (Light Detection and Ranging) sensors use laser light to create 3D maps of their surroundings.

   - Suitable for vehicle and pedestrian detection and tracking.

5. Video Cameras:

   - Video cameras capture real-time images and video footage of traffic.

   - Used for visual analysis, license plate recognition, and incident detection.

   - Implement computer vision algorithms for vehicle and object recognition.

6. Thermal Cameras:

   - Thermal cameras detect heat signatures, making them useful for low-light and adverse weather conditions.

   - Can detect pedestrians and vehicles based on temperature differences.

7. Environmental Sensors:

   - Environmental sensors measure weather conditions such as temperature, humidity, and visibility.

   - Data from these sensors can provide insights into how weather affects traffic.

8. Vehicle-to-Infrastructure (V2I) Communication Devices:

   - V2I devices enable direct communication between vehicles and infrastructure.

   - They can provide information to vehicles about traffic conditions and traffic signal timings.

9. GPS Devices:

   - GPS modules in vehicles can transmit real-time location data.

   - Used for traffic monitoring and routing optimization.

10. Microcontrollers and IoT Gateways:

   - These devices act as the interface between sensors and the central server.

   - They collect data from sensors and transmit it to the server using communication protocols such as MQTT or HTTP.

11. Power and Connectivity Solutions:

   - Ensure power sources for IoT devices, such as solar panels or wired power.

   - Establish reliable connectivity options, such as cellular, Wi-Fi, or Ethernet.

12. Data Aggregation and Transmission:

   - Data collected from sensors and cameras is aggregated and transmitted to a central server for processing and analysis.

13. Data Security and Privacy:

   - Implement security measures to protect data integrity and privacy, especially for video and sensor data.

14. Maintenance and Calibration:

   - Regular maintenance and calibration of sensors and cameras to ensure accurate data collection.

When implementing IoT sensors and cameras in a Traffic Management System, consider the specific needs of your project, the traffic patterns in the area, and the desired level of detail and accuracy. The choice of sensors and cameras will depend on factors like cost, environmental conditions, and the types of data you need for traffic management and analysis.

|  |  |
| --- | --- |
|  |  |

**B. Data Transmission:**

Data transmission in IoT (Internet of Things) involves sending and receiving data between IoT devices, sensors, and the central processing systems. Here's an overview of how it works:

1. Data Generation:

IoT devices, such as sensors, collect data from their surroundings. This data can include temperature, humidity, motion, or any other relevant information.

2. Data Processing:

Some IoT devices have limited processing capabilities, while others preprocess data locally. They may filter, aggregate, or format the data before transmission.

3. Data Transmission:

IoT devices transmit data using various communication protocols, including Wi-Fi, cellular, Bluetooth, LoRa, Zigbee, or even satellite communication. The choice of protocol depends on the application, range, power constraints, and data volume.

4. Data Aggregation:

In some cases, data from multiple IoT devices may be aggregated at a local gateway before being sent to a central server or cloud platform. This helps reduce the load on individual devices and enables more efficient data transmission.

5. Cloud or Edge Processing:

Data is received by a central server in the cloud or processed at the edge, depending on the architecture. Edge computing allows for real-time processing and faster response times, while cloud computing provides scalability and extensive data analytics.

6. Data Storage and Analysis:

The received data is stored in databases and analyzed for insights. Machine learning and analytics algorithms can be applied to make sense of the data and trigger automated actions.

7. Feedback and Control:

In some cases, the central system may send commands or feedback to IoT devices, enabling remote control or adjustments based on the data analysis.

Security is a critical aspect of IoT data transmission. Encryption, authentication, and access control are implemented to protect the integrity and confidentiality of the data.

Remember that the specific implementation of data transmission in IoT can vary widely based on the application, devices, and network infrastructure involved.

|  |  |
| --- | --- |
|  |  |

**C. Central Server:**

A central server in an IoT (Internet of Things) system plays a crucial role in managing, processing, and controlling IoT devices and their data. Here's an overview of the central server's functions and its role within an IoT ecosystem:

1. Data Collection and Aggregation:

The central server collects data from various IoT devices and sensors. This data can be generated from different locations and can include information like temperature, humidity, sensor readings, and more. The central server aggregates this data for processing.

2. Data Storage:

The collected data is typically stored in databases, often on the cloud. This data repository allows for historical analysis, real-time access, and ensures data integrity and redundancy.

3. Data Processing and Analytics:

The central server processes the IoT data, performing various operations such as filtering, analysis, and running machine learning algorithms. This enables insights, trends, and actionable information to be derived from the raw data.

4. Remote Device Management:

The central server can send commands to IoT devices for remote management and control. It allows you to update device settings, trigger actions, or even perform over-the-air firmware updates.

5. User Interface:

A central server often provides a web-based or mobile application interface for users or administrators. This interface allows them to monitor and control IoT devices, access data, set alerts, and make informed decisions based on the IoT data.

6. Security:

Ensuring the security of IoT data and device communication is a critical role of the central server. It implements access control, authentication, and encryption to protect the data and maintain the integrity of the system.

7. Scalability:

Central servers are designed to be scalable to handle a growing number of IoT devices and data. They can adapt to changing requirements and accommodate additional devices seamlessly.

8. Integration:

Central servers often have the capability to integrate with other systems and platforms. This can include integration with external services, APIs, or other enterprise systems to enhance functionality and data exchange.

9. Alerts and Notifications:

The central server can be configured to send alerts and notifications based on predefined conditions. For example, it can alert users or administrators if a sensor detects an anomaly or a threshold is breached.

10. Data Sharing:

Depending on the application, the central server can facilitate data sharing with third parties or external stakeholders, allowing them to access specific data or insights.

The design and implementation of a central server for IoT can vary greatly based on the specific requirements of the IoT application, the number and types of devices, and the desired features and capabilities. It's a central component that orchestrates the functioning of the entire IoT ecosystem.

**D.** **Data Processing and Analysis:**

   Data processing and analysis are fundamental components of IoT (Internet of Things) systems. Here's how data processing and analysis work within the context of IoT:

1. Data Collection:

IoT devices, such as sensors and actuators, generate data based on the environment they are monitoring. This data can include temperature readings, motion detection, humidity levels, and much more.

2. Data Transmission:

IoT devices transmit this data to a central server or cloud platform through various communication protocols. The data can be sent in real-time or at scheduled intervals, depending on the application.

3. Data Preprocessing:

Upon receiving the data, preprocessing steps may be applied. This can involve filtering out noise, converting data formats, or resampling to reduce redundancy.

4. Data Storage:

The processed data is then stored in databases, often leveraging scalable and distributed data storage solutions. This ensures that the data is accessible for historical analysis and future reference.

5. Data Analysis:

   - Real-time Analysis: Some IoT applications require real-time data analysis, where data is processed and acted upon immediately. For example, in a smart home, IoT devices can trigger actions like turning on lights when motion is detected.

   - Batch Analysis: Historical data can be analyzed in batches to identify long-term trends, anomalies, or patterns. This can involve running analytics and machine learning algorithms.

   - Predictive Analysis: IoT data can be used to build predictive models. For instance, predictive maintenance for industrial equipment can use IoT data to forecast when maintenance is needed, reducing downtime.

   - Anomaly Detection: Data analysis can identify anomalies or deviations from expected patterns, which may indicate issues or security breaches.

   - Data Correlation: Data from various IoT devices can be correlated to gain more comprehensive insights. For example, correlating weather data with energy consumption in a smart building.

6. Visualization:

The results of data analysis can be presented through visual dashboards and reports. Visualization tools help users understand the data and make informed decisions. Charts, graphs, and alerts can be used to convey the findings.

7. Actionable Insights:

The goal of data processing and analysis is to derive actionable insights. For example, in agriculture, IoT data can be used to optimize irrigation schedules, leading to water savings and increased crop yields.

8. Automation:

In some cases, IoT systems can be designed to trigger automated actions based on the data analysis. For example, IoT devices can automatically adjust room temperature based on occupancy and weather conditions in a smart building.

9. Scalability:

IoT data processing and analysis should be scalable to handle increasing volumes of data as more devices are added to the network.

10. Security:

Security is paramount in data processing and analysis in IoT to protect the integrity and confidentiality of the data. Encryption, access control, and authentication measures are implemented.

IoT data processing and analysis can occur either at the edge (on the IoT device itself), in a local gateway, or in the cloud, depending on the application's requirements and latency constraints. The choice of data processing and analysis techniques depends on the specific use case and the desired outcomes.

|  |  |
| --- | --- |
|  |  |

**E. Traffic Control and Optimization:**

Traffic control and optimization using IoT (Internet of Things) involves leveraging IoT devices, sensors, and data analysis to manage and improve traffic flow, reduce congestion, enhance safety, and provide more efficient transportation systems. Here's how it works:

1. Traffic Sensors:

IoT sensors are deployed at various locations in the road network. These sensors can include cameras, radar detectors, ultrasonic sensors, and even smartphones. They collect data on traffic conditions, vehicle speed, and traffic density.

2. Data Collection:

The sensors continuously collect data and transmit it to a central server or cloud platform. This data includes real-time information about traffic flow, congestion, accidents, and road conditions.

3. Data Analysis:

The collected data is processed and analyzed in real-time to extract meaningful insights. Algorithms can detect traffic congestion, identify accident-prone areas, and predict traffic patterns based on historical data.

4. Traffic Management:

   - Dynamic Traffic Signals: Traffic signals can be adjusted in real-time based on traffic conditions. For instance, green light durations can be extended in the direction with heavier traffic.

   - Adaptive Traffic Control: Traffic control systems can adapt to the ebb and flow of traffic, optimizing signal timing to reduce waiting times and congestion.

   - Traffic Diversion: Based on real-time data, systems can suggest alternate routes to divert traffic away from congested areas.

   - Emergency Response: In the event of accidents or emergencies, IoT systems can alert authorities and reroute traffic to ensure quick emergency response.

5. Public Information:

IoT-based traffic systems can communicate real-time traffic information to drivers and commuters through digital signs, mobile apps, or vehicle navigation systems. This helps drivers make informed decisions about their routes.

6. Parking Optimization:

IoT sensors can be used to monitor available parking spaces in urban areas. Drivers can be directed to vacant parking spots, reducing traffic caused by people searching for parking.

7. Predictive Maintenance:

IoT sensors on road infrastructure can monitor the condition of roads and infrastructure elements like bridges. They can predict when maintenance is needed to prevent road closures and accidents.

8. Environmental Impact:

IoT can help monitor and reduce the environmental impact of traffic by optimizing routes to reduce fuel consumption and emissions.

9. Data Sharing:

Governments, transportation authorities, and third-party developers can access the traffic data for research, urban planning, and the development of new transportation solutions.

10. Security:

Ensuring the security of IoT devices and data is crucial in traffic control systems. Protection against cyber threats and unauthorized access is a priority.

By harnessing IoT for traffic control and optimization, cities and transportation authorities can significantly improve traffic management, reduce congestion, save time and fuel, enhance road safety, and make urban mobility more efficient and sustainable.

|  |  |
| --- | --- |
|  |  |

**F. Dashboard and User Interface:**

 Creating a dashboard and user interface for IoT (Internet of Things) devices typically involves several steps:

1. Define Your Requirements:

Determine what data you want to display, the user interactions you need, and the devices you're connecting. This will guide your design.

2. Choose a Platform:

Select a platform or framework for building the dashboard. Some popular choices include IoT platforms like AWS IoT, Google Cloud IoT, or open-source solutions like Node-RED.

3. Data Collection:

Connect your IoT devices to the platform and set up data collection. Ensure your devices are sending data to the platform in a format you can work with.

4. User Interface Design:

Design the user interface of the dashboard. Consider the layout, visual elements, and how users will interact with it. Tools like HTML, CSS, and JavaScript can be used for web-based interfaces.

5. Data Visualization:

Create visual representations of the IoT data. Charts, graphs, and tables are common elements to display information effectively.

6. Real-Time Updates:

If needed, make sure your dashboard can update in real-time to reflect the latest data from your IoT devices.

7. Security:

Implement strong security measures to protect data and the devices themselves. Use encryption, authentication, and authorization mechanisms.

8. User Authentication:

Implement user authentication to ensure only authorized users can access the dashboard.

9. Testing:

Thoroughly test the dashboard and interface to ensure it functions correctly and is user-friendly.

10. Deployment:

Deploy the dashboard to a web server or a platform that can be accessed by users.

11. Monitoring and Maintenance:

Continuously monitor the dashboard and IoT devices for issues. Regularly update the interface and security measures.

12. User Training:

If necessary, provide training to users on how to use the dashboard effectively.

Remember that the specific technologies and tools you use will depend on your project's requirements, but the above steps provide a general roadmap for creating a dashboard and user interface for IoT.

|  |  |
| --- | --- |
|  |  |

**This project involves a significant amount of hardware, software, and infrastructure development. Depending on your resources and requirements, you may want to work on specific aspects of the system first before attempting to build the complete solution**.

**Remember to comply with local regulations and consider collaboration with local authorities for real-world deployments of a traffic management system.**

|  |  |
| --- | --- |
|  |  |

#import lib files

from tracking.centroidtracker import CentroidTracker

from tracking.trackableobject import TrackableObject

import tensornets as nets

import cv2

import numpy as np

import time

import dlib

import tensorflow.compat.v1 as tf

import os

import threading

def countVehicles(param):

# param -> path of the video

# list -> number of vehicles will be written in the list

# index ->Index at which data has to be written

tf.disable\_v2\_behavior()

# Image size must be '416x416' as YoloV3 network expects that specific image size as input

img\_size = 416

    inputs = tf.placeholder(tf.float32, [None, img\_size, img\_size, 3])

    model = nets.YOLOv3COCO(inputs, nets.Darknet19)

ct = CentroidTracker(maxDisappeared=5, maxDistance=50) # Look into 'CentroidTracker' for further info about parameters

    trackers = [] # List of all dlib trackers

    trackableObjects = {} # Dictionary of trackable objects containing object's ID and its' corresponding centroid/s

    skip\_frames = 10 # Numbers of frames to skip from detecting

    confidence\_level = 0.40 # The confidence level of a detection

    total = 0 # Total number of detected objects from classes of interest

    use\_original\_video\_size\_as\_output\_size = True # Shows original video as output and not the 416x416 image that is used as yolov3 input (NOTE: Detection still happens with 416x416 img size but the output is displayed in original video size if this parameter is True)

    video\_path = os.getcwd() + param # "/videos/4.mp4"

    video\_name = os.path.basename(video\_path)

# print("Loading video {video\_path}...".format(video\_path=video\_path))

if not os.path.exists(video\_path):

        print("File does not exist. Exited.")

        exit()

# YoloV3 detects 80 classes represented below

all\_classes = ["person", "bicycle", "car", "motorbike", "aeroplane", "bus", "train", "truck", \

                  "boat", "traffic light", "fire hydrant", "stop sign", "parking meter", "bench", \

                  "bird", "cat", "dog", "horse", "sheep", "cow", "elephant", "bear", "zebra", "giraffe", \

                  "backpack", "umbrella", "handbag", "tie", "suitcase", "frisbee", "skis", "snowboard", \

                  "sports ball", "kite", "baseball bat", "baseball glove", "skateboard", "surfboard", \

                  "tennis racket", "bottle", "wine glass", "cup", "fork", "knife", "spoon", "bowl", "banana", \

                  "apple", "sandwich", "orange", "broccoli", "carrot", "hot dog", "pizza", "donut", "cake", \

                  "chair", "sofa", "pottedplant", "bed", "diningtable", "toilet", "tvmonitor", "laptop", "mouse", \

                  "remote", "keyboard", "cell phone", "microwave", "oven", "toaster", "sink", "refrigerator", \

                  "book", "clock", "vase", "scissors", "teddy bear", "hair drier", "toothbrush"]

# Classes of interest (with their corresponding indexes for easier looping)

classes = { 1 : 'bicycle', 2 : 'car', 3 : 'motorbike', 5 : 'bus', 7 : 'truck' }

    with tf.Session() as sess:

        sess.run(model.pretrained())

        cap = cv2.VideoCapture(video\_path)

# Get video size (just for log purposes)

width =  int(cap.get(cv2.CAP\_PROP\_FRAME\_WIDTH))

        height = int(cap.get(cv2.CAP\_PROP\_FRAME\_HEIGHT))

# Scale used for output window size and net size

width\_scale = 1

        height\_scale = 1

        if use\_original\_video\_size\_as\_output\_size:

            width\_scale = width / img\_size

            height\_scale = height / img\_size

        def drawRectangleCV2(img, pt1, pt2, color, thickness, width\_scale=width\_scale, height\_scale=height\_scale):

            point1 = (int(pt1[0] \* width\_scale), int(pt1[1] \* height\_scale))

            point2 = (int(pt2[0] \* width\_scale), int(pt2[1] \* height\_scale))

            return cv2.rectangle(img, point1, point2, color, thickness)

        def drawTextCV2(img, text, pt, font, font\_scale, color, lineType, width\_scale=width\_scale, height\_scale=height\_scale):

            pt = (int(pt[0] \* width\_scale), int(pt[1] \* height\_scale))

            cv2.putText(img, text, pt, font, font\_scale, color, lineType)

        def drawCircleCV2(img, center, radius, color, thickness, width\_scale=width\_scale, height\_scale=height\_scale):

            center = (int(center[0] \* width\_scale), int(center[1] \* height\_scale))

            cv2.circle(img, center, radius, color, thickness)

# Python 3.5.6 does not support f-strings (next line will generate syntax error)

#print(f"Loaded {video\_path}. Width: {width}, Height: {height}")

# print("Loaded {video\_path}. Width: {width}, Height: {height}".format(video\_path=video\_path, width=width, height=height))

skipped\_frames\_counter = 0

        while(cap.isOpened()):

            try :

                ret, frame = cap.read()

                img = cv2.resize(frame, (img\_size, img\_size))

            except:

                print(total\_str)

            output\_img = frame if use\_original\_video\_size\_as\_output\_size else img

            tracker\_rects = []

            if skipped\_frames\_counter == skip\_frames:

# Detecting happens after number of frames have passes specified by 'skip\_frames' variable value

# print("[DETECTING]")

trackers = []

                skipped\_frames\_counter = 0 # reset counter

                np\_img = np.array(img).reshape(-1, img\_size, img\_size, 3)

                start\_time=time.time()

                predictions = sess.run(model.preds, {inputs: model.preprocess(np\_img)})

# print("Detection took %s seconds" % (time.time() - start\_time))

# model.get\_boxes returns a 80 element array containing information about detected classes

# each element contains a list of detected boxes, confidence level ...

detections = model.get\_boxes(predictions, np\_img.shape[1:3])

                np\_detections = np.array(detections)

# Loop only through classes we are interested in

for class\_index in classes.keys():

                    local\_count = 0

                    class\_name = classes[class\_index]

# Loop through detected infos of a class we are interested in

for i in range(len(np\_detections[class\_index])):

                        box = np\_detections[class\_index][i]

                        if np\_detections[class\_index][i][4] >= confidence\_level:

# print("Detected ", class\_name, " with confidence of ", np\_detections[class\_index][i][4])

local\_count += 1

                            startX, startY, endX, endY = box[0], box[1], box[2], box[3]

                            drawRectangleCV2(output\_img, (startX, startY), (endX, endY), (0, 255, 0), 1)

                            drawTextCV2(output\_img, class\_name, (startX, startY), cv2.FONT\_HERSHEY\_SIMPLEX, .5, (0, 0, 255), 1)

# Construct a dlib rectangle object from the bounding box coordinates and then start the dlib correlation

tracker = dlib.correlation\_tracker()

                            rect = dlib.rectangle(int(startX), int(startY), int(endX), int(endY))

                            tracker.start\_track(img, rect)

# Add the tracker to our list of trackers so we can utilize it during skip frames

# Write the total number of detected objects for a given class on this frame

# print(class\_name," : ", local\_count)

else:

# If detection is not happening then track previously detected objects (if any)

# print("[TRACKING]")

skipped\_frames\_counter += 1

# Increase the number frames for which we did not use detection

# Loop through tracker, update each of them and display their rectangle

for tracker in trackers:

                    tracker.update(img)

                    pos = tracker.get\_position()

# Unpack the position object

startX = int(pos.left())

                    startY = int(pos.top())

                    endX = int(pos.right())

                    endY = int(pos.bottom())

# Add the bounding box coordinates to the tracking rectangles list

                    tracker\_rects.append((startX, startY, endX, endY))

# Draw tracking rectangles

                    drawRectangleCV2(output\_img, (startX, startY), (endX, endY), (255, 0, 0), 1)

# Use the centroid tracker to associate the (1) old object centroids with (2) the newly computed object centroids

            objects = ct.update(tracker\_rects)

# Loop over the tracked objects

            for (objectID, centroid) in objects.items():

# Check to see if a trackable object exists for the current object ID

to = trackableObjects.get(objectID, None)

                if to is None:

# If there is no existing trackable object, create one

to = TrackableObject(objectID, centroid)

                else:

                    to.centroids.append(centroid)

# If the object has not been counted, count it and mark it as counted

if not to.counted:

                        total += 1

                        to.counted = True

# Store the trackable object in our dictionary

                trackableObjects[objectID] = to

# Draw both the ID of the object and the centroid of the object on the output frame

object\_id = "ID {}".format(objectID)

                drawTextCV2(output\_img, object\_id, (centroid[0] - 10, centroid[1] - 10), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (0, 255, 0), 1)

                drawCircleCV2(output\_img, (centroid[0], centroid[1]), 2, (0, 255, 0), -1)

# Display the total count so far

total\_str = str(total)

                drawTextCV2(output\_img, total\_str, (10, 30), cv2.FONT\_HERSHEY\_SIMPLEX, 0.6, (0, 0, 255), 2)

# Display the current frame (with all annotations drawn up to this point)

cv2.imshow(video\_name, output\_img)

            key = cv2.waitKey(1) & 0xFF

            if key  == ord('q'): # QUIT (exits)

                break

            elif key == ord('p'):

                cv2.waitKey(0) # PAUSE (Enter any key to continue)

    cap.release()

    cv2.destroyAllWindows()

    print("Exited")

    """

    function which will run our code

    will write the number of veicles in the list provided

    """

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

    countVehicles("/videos/test.mp4")

# Logic for setting the time for each signal

A Traffic Management System (TMS) typically consists of various components and subsystems that work

together to manage and control traffic flow. Here's a simplified block diagram explanation of a TMS:

1. \*Traffic Sensors\*: These are various sensors placed on the road, such as cameras, inductive loops, or

radar detectors. They collect data on traffic conditions, including vehicle presence, speed, and volume.

2. \*Data Collection and Processing\*: The data from the sensors are transmitted to a central control

system. This system collects, processes, and analyzes the data to understand the current traffic

situation.

3. \*Traffic Control Center\*: This is where traffic controllers and operators monitor the data from the

sensors in real-time. They use this information to make decisions and control traffic signals, variable

message signs, and other devices.

4. \*Traffic Signals and Signs\*: Traffic signals at intersections can be controlled in real-time based on

traffic conditions. Variable message signs along roads can display information and warnings to drivers.

5. \*Communication Network\*: A reliable communication network, often using fiber optics or wireless

technology, connects the various components of the system, ensuring data is transmitted quickly and

without interruption.

6. \*Traffic Management Algorithms\*: These are the software algorithms that process the data and make

decisions about traffic signal timing, lane control, and other actions to optimize traffic flow and safety.

7. \*Emergency Response Integration\*: TMS can be integrated with emergency services to ensure rapid

response to incidents, such as accidents or road closures.

8. \*Driver Information Systems\*: TMS can provide real-time information to drivers through apps,

websites, or electronic signs, helping them make informed decisions about their routes.

9. \*Traffic Data Storage and Analysis\*: Historical traffic data can be stored and analyzed for long-term

planning and optimization of the road network.

10. \*Maintenance and Diagnostics\*: Systems for monitoring and maintaining the sensors, control

equipment, and other components to ensure they are functioning correctly.

11. \*Feedback Loop\*: Continuous monitoring and feedback from the system allow for adjustments and

improvements to traffic management strategies.

This block diagram shows the major components of a Traffic Management System, which collectively

work to enhance traffic safety, reduce congestion, and improve the overall efficiency of road networks.

The specifics of each TMS can vary depending on the scale and complexity of the transportation system

it serves.

#import lib files

from tracking.centroidtracker import CentroidTracker

from tracking.trackableobject import TrackableObject

import tensornets as nets

import cv2

import numpy as np

import time

import dlib

import tensorflow.compat.v1 as tf

import os

import threading

def countVehicles(param):

# param -> path of the video

# list -> number of vehicles will be written in the list

# index ->Index at which data has to be written

tf.disable\_v2\_behavior()

# Image size must be '416x416' as YoloV3 network expects that specific image size as input

img\_size = 416

inputs = tf.placeholder(tf.float32, [None, img\_size, img\_size, 3])

model = nets.YOLOv3COCO(inputs, nets.Darknet19)

ct = CentroidTracker(maxDisappeared=5, maxDistance=50) # Look into 'CentroidTracker' for further info

about parameters

trackers = [] # List of all dlib trackers

trackableObjects = {} # Dictionary of trackable objects containing object's ID and its' corresponding

centroid/s

skip\_frames = 10 # Numbers of frames to skip from detecting

confidence\_level = 0.40 # The confidence level of a detection

total = 0 # Total number of detected objects from classes of interest

use\_original\_video\_size\_as\_output\_size = True # Shows original video as output and not the 416x416

image that is used as yolov3 input (NOTE: Detection still happens with 416x416 img size but the output

is displayed in original video size if this parameter is True)

video\_path = os.getcwd() + param # "/videos/4.mp4"

video\_name = os.path.basename(video\_path)

# print("Loading video {video\_path}...".format(video\_path=video\_path))

if not os.path.exists(video\_path):

print("File does not exist. Exited.")

exit()

# YoloV3 detects 80 classes represented below

all\_classes = ["person", "bicycle", "car", "motorbike", "aeroplane", "bus", "train", "truck", \

"boat", "traffic light", "fire hydrant", "stop sign", "parking meter", "bench", \

"bird", "cat", "dog", "horse", "sheep", "cow", "elephant", "bear", "zebra", "giraffe", \

"backpack", "umbrella", "handbag", "tie", "suitcase", "frisbee", "skis", "snowboard", \

"sports ball", "kite", "baseball bat", "baseball glove", "skateboard", "surfboard", \

"tennis racket", "bottle", "wine glass", "cup", "fork", "knife", "spoon", "bowl", "banana", \

"apple", "sandwich", "orange", "broccoli", "carrot", "hot dog", "pizza", "donut", "cake", \

"chair", "sofa", "pottedplant", "bed", "diningtable", "toilet", "tvmonitor", "laptop", "mouse", \

"remote", "keyboard", "cell phone", "microwave", "oven", "toaster", "sink", "refrigerator", \

"book", "clock", "vase", "scissors", "teddy bear", "hair drier", "toothbrush"]

# Classes of interest (with their corresponding indexes for easier looping)

classes = { 1 : 'bicycle', 2 : 'car', 3 : 'motorbike', 5 : 'bus', 7 : 'truck' }

with tf.Session() as sess:

sess.run(model.pretrained())

cap = cv2.VideoCapture(video\_path)

# Get video size (just for log purposes)

width = int(cap.get(cv2.CAP\_PROP\_FRAME\_WIDTH))

height = int(cap.get(cv2.CAP\_PROP\_FRAME\_HEIGHT))

# Scale used for output window size and net size

width\_scale = 1

height\_scale = 1

if use\_original\_video\_size\_as\_output\_size:

width\_scale = width / img\_size

height\_scale = height / img\_size

def drawRectangleCV2(img, pt1, pt2, color, thickness, width\_scale=width\_scale,

height\_scale=height\_scale):

point1 = (int(pt1[0] \* width\_scale), int(pt1[1] \* height\_scale))

point2 = (int(pt2[0] \* width\_scale), int(pt2[1] \* height\_scale))

return cv2.rectangle(img, point1, point2, color, thickness)

def drawTextCV2(img, text, pt, font, font\_scale, color, lineType, width\_scale=width\_scale,

height\_scale=height\_scale):

pt = (int(pt[0] \* width\_scale), int(pt[1] \* height\_scale))

cv2.putText(img, text, pt, font, font\_scale, color, lineType)

def drawCircleCV2(img, center, radius, color, thickness, width\_scale=width\_scale,

height\_scale=height\_scale):

center = (int(center[0] \* width\_scale), int(center[1] \* height\_scale))

cv2.circle(img, center, radius, color, thickness)

# Python 3.5.6 does not support f-strings (next line will generate syntax error)

#print(f"Loaded {video\_path}. Width: {width}, Height: {height}")

# print("Loaded {video\_path}. Width: {width}, Height:

{height}".format(video\_path=video\_path, width=width, height=height))

skipped\_frames\_counter = 0

while(cap.isOpened()):

try :

ret, frame = cap.read()

img = cv2.resize(frame, (img\_size, img\_size))

except:

print(total\_str)

output\_img = frame if use\_original\_video\_size\_as\_output\_size else img

tracker\_rects = []

if skipped\_frames\_counter == skip\_frames:

# Detecting happens after number of frames have passes specified by 'skip\_frames' variable value

# print("[DETECTING]")

trackers = []

skipped\_frames\_counter = 0 # reset counter

np\_img = np.array(img).reshape(-1, img\_size, img\_size, 3)

start\_time=time.time()

predictions = sess.run(model.preds, {inputs: model.preprocess(np\_img)})

# print("Detection took %s seconds" % (time.time() - start\_time))

# model.get\_boxes returns a 80 element array containing information

about detected classes

# each element contains a list of detected boxes, confidence level ...

detections = model.get\_boxes(predictions, np\_img.shape[1:3])

np\_detections = np.array(detections)

# Loop only through classes we are interested in

for class\_index in classes.keys():

local\_count = 0

class\_name = classes[class\_index]

# Loop through detected infos of a class we are interested in

for i in range(len(np\_detections[class\_index])):

box = np\_detections[class\_index][i]

if np\_detections[class\_index][i][4] >= confidence\_level:

# print("Detected ", class\_name, " with confidence of ", np\_detections[class\_index][i][4])

local\_count += 1

startX, startY, endX, endY = box[0], box[1], box[2], box[3]

drawRectangleCV2(output\_img, (startX, startY), (endX, endY), (0, 255, 0), 1)

drawTextCV2(output\_img, class\_name, (startX, startY), cv2.FONT\_HERSHEY\_SIMPLEX, .5,

(0, 0, 255), 1)

# Construct a dlib rectangle object from the

bounding box coordinates and then start the dlib correlation

tracker = dlib.correlation\_tracker()

rect = dlib.rectangle(int(startX), int(startY), int(endX), int(endY))

tracker.start\_track(img, rect)

# Add the tracker to our list of trackers so we

can utilize it during skip frames

# Write the total number of detected objects for a given class on this frame

# print(class\_name," : ", local\_count)

else:

# If detection is not happening then track previously detected objects (if any)

# print("[TRACKING]")

skipped\_frames\_counter += 1

# Increase the number frames for which we did not use detection

# Loop through tracker, update each of them and display their rectangle

for tracker in trackers:

tracker.update(img)

pos = tracker.get\_position()

# Unpack the position object

startX = int(pos.left())

startY = int(pos.top())

endX = int(pos.right())

endY = int(pos.bottom())

# Add the bounding box coordinates to the tracking rectangles

list

tracker\_rects.append((startX, startY, endX, endY))

# Draw tracking rectangles

drawRectangleCV2(output\_img, (startX, startY), (endX, endY), (255, 0, 0), 1)

# Use the centroid tracker to associate the (1) old object centroids with (2) the

newly computed object centroids

objects = ct.update(tracker\_rects)

# Loop over the tracked objects

for (objectID, centroid) in objects.items():

# Check to see if a trackable object exists for the current object ID

to = trackableObjects.get(objectID, None)

if to is None:

# If there is no existing trackable object, create one

to = TrackableObject(objectID, centroid)

else:

to.centroids.append(centroid)

# If the object has not been counted, count it and mark it as

counted

if not to.counted:

total += 1

to.counted = True

# Store the trackable object in our dictionary

trackableObjects[objectID] = to

# Draw both the ID of the object and the centroid of the object on the

output frame

object\_id = "ID {}".format(objectID)

drawTextCV2(output\_img, object\_id, (centroid[0] - 10, centroid[1] - 10),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (0, 255, 0), 1)

drawCircleCV2(output\_img, (centroid[0], centroid[1]), 2, (0, 255, 0), -1)

# Display the total count so far

total\_str = str(total)

drawTextCV2(output\_img, total\_str, (10, 30), cv2.FONT\_HERSHEY\_SIMPLEX, 0.6, (0, 0, 255), 2)

# Display the current frame (with all annotations drawn up to this point)

cv2.imshow(video\_name, output\_img)

key = cv2.waitKey(1) & 0xFF

if key == ord('q'): # QUIT (exits)

break

elif key == ord('p'):

cv2.waitKey(0) # PAUSE (Enter any key to continue)

cap.release()

cv2.destroyAllWindows()

print("Exited")

"""

function which will run our code

will write the number of veicles in the list provided

"""

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

countVehicles("/videos/test.mp4")

# Logic for setting the time for each signal