**TRAFFIC MANAGEMENT SYSTEM**

**SOFTWARE REQUIREMENTS:**

The essential software components used in building a Traffic Management System (TMS) include:

* Traffic data collection and analysis software.
* Traffic signal control software.
* Incident management software.
* Data sharing and communication software.
* Predictive modelling software.
* Public transportation integration software.
* Environmental impact management software.
* Remote monitoring and control software.

**HARDWARE REQUIREMENTS:**

For a project integrating historical traffic data and machine learning for congestion prediction using IoT, the hardware requirements include:

* Traffic sensors
* IoT gateways
* Data processing servers
* IoT network infrastructure
* Machine learning hardware
* Data storage
* Power backup systems
* Traffic cameras

**DESIGN FLOW:**

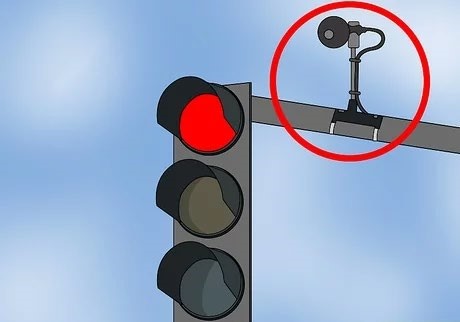
The project flow for integrating historical traffic data and machine learning for congestion prediction

using IoT can be distilled into key steps:

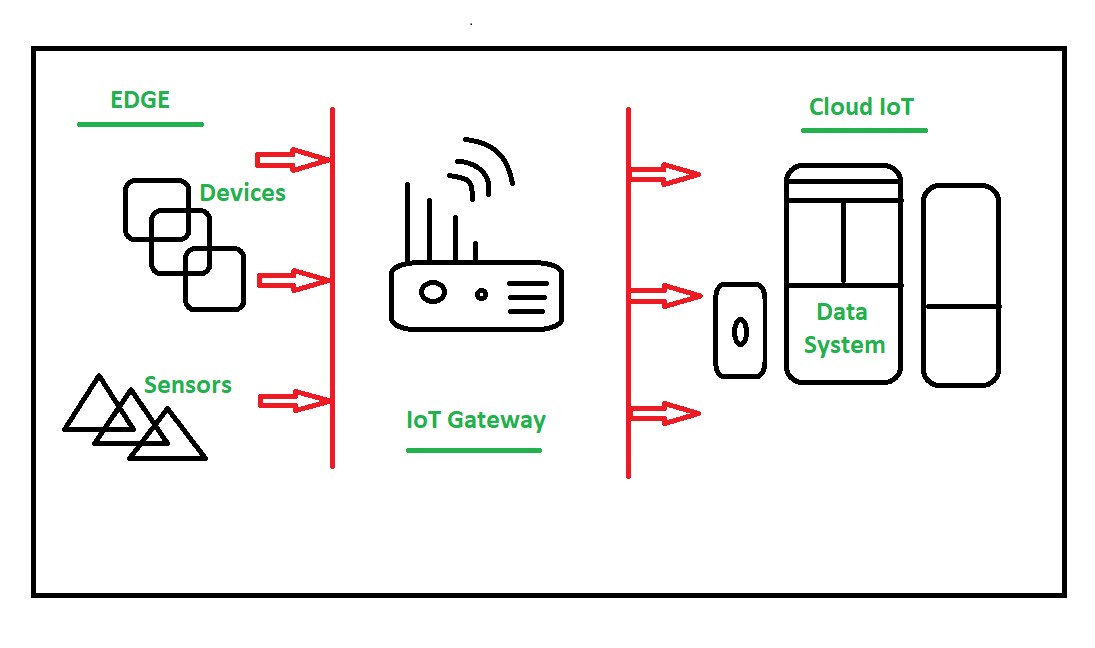
* Planning and Hardware Setup: Define project goals and select hardware components.
* Data Collection: Gather real-time data from sensors and IoT gateways.
* Data Processing and Storage: Process and store data for analysis.
* Machine Learning Models: Develop predictive models based on historical data.
* Real-time Analysis: Continuously analyze current traffic data.
* Congestion Prediction: Use machine learning to forecast congestion.
* Data Visualization and Reporting: Present insights through user interfaces and reports.
* Deployment and Maintenance: Launch the system and maintain its performance.
* Security and Compliance: Ensure data and system security, and adhere to regulations.
* Testing, Training, and Scalability: Rigorously test the system, train users, and plan for future growth.

**COMPONENTS:**

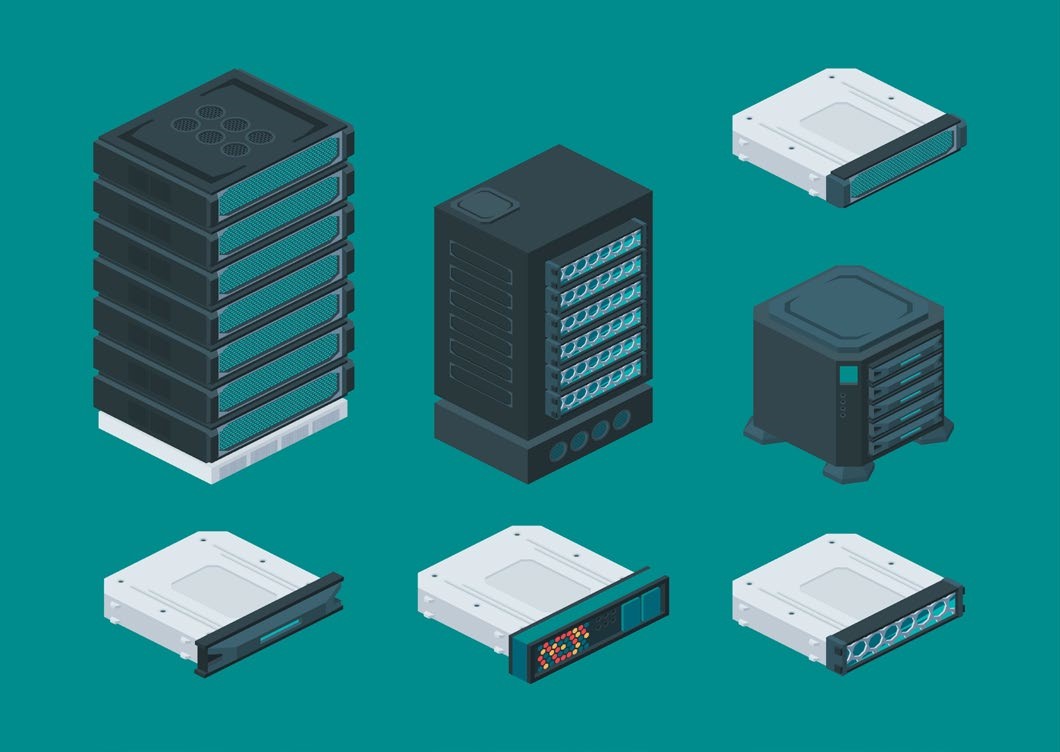
The following are the selected hardware components for the project:



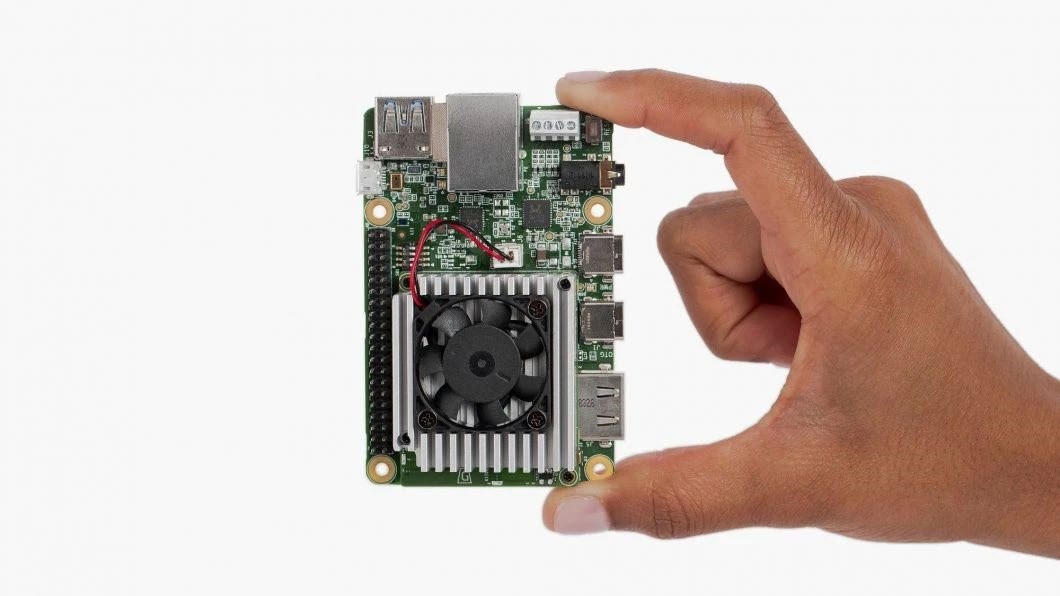
**Traffic Sensors:** These are various sensors such as loop detectors, radar sensors, and cameras used to collect real-time traffic data.



**IoT Gateways:** Devices that collect data from traffic sensors and transmit it to a central processing system.



**Data Processing Servers:** High-performance servers for data storage, pre-processing, and running machine learning algorithms.

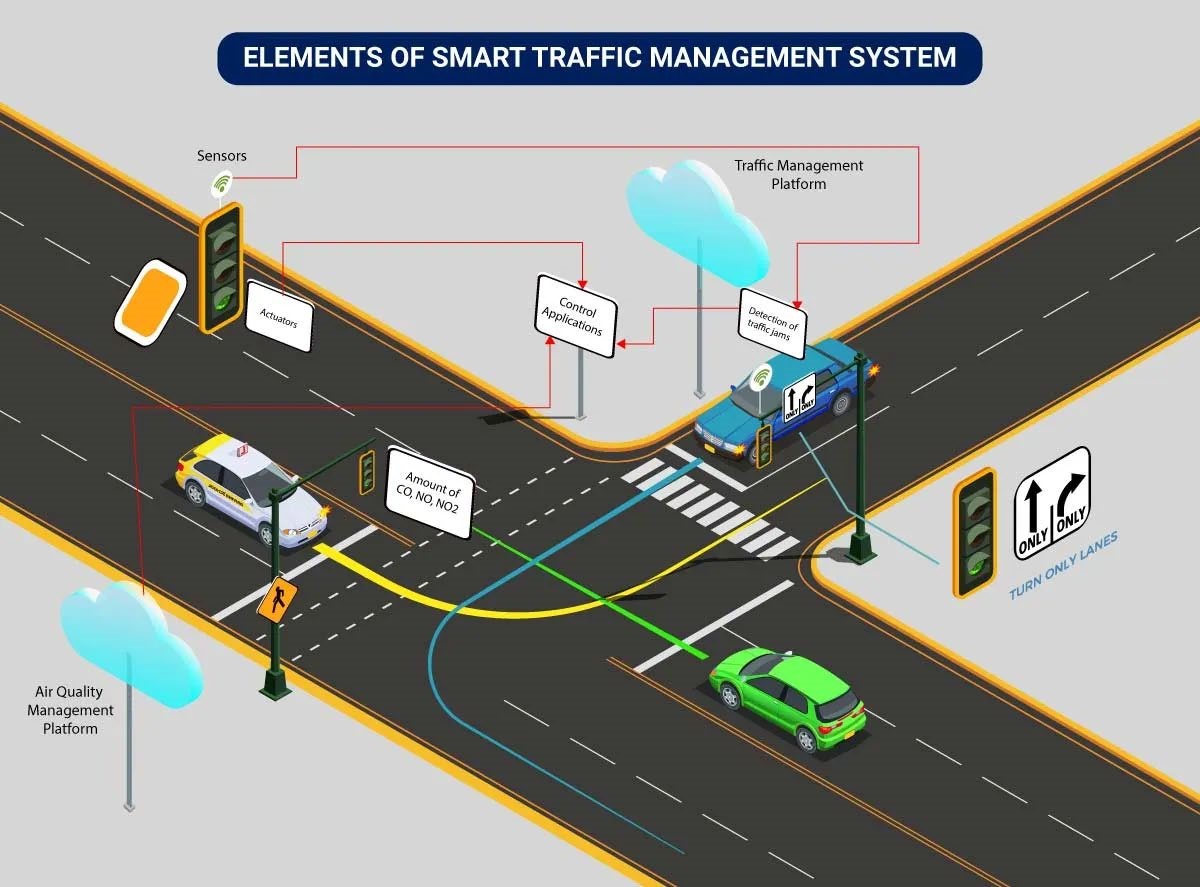


**Machine Learning Hardware:** Hardware accelerators like GPUs or TPUs used for machine learning model training.

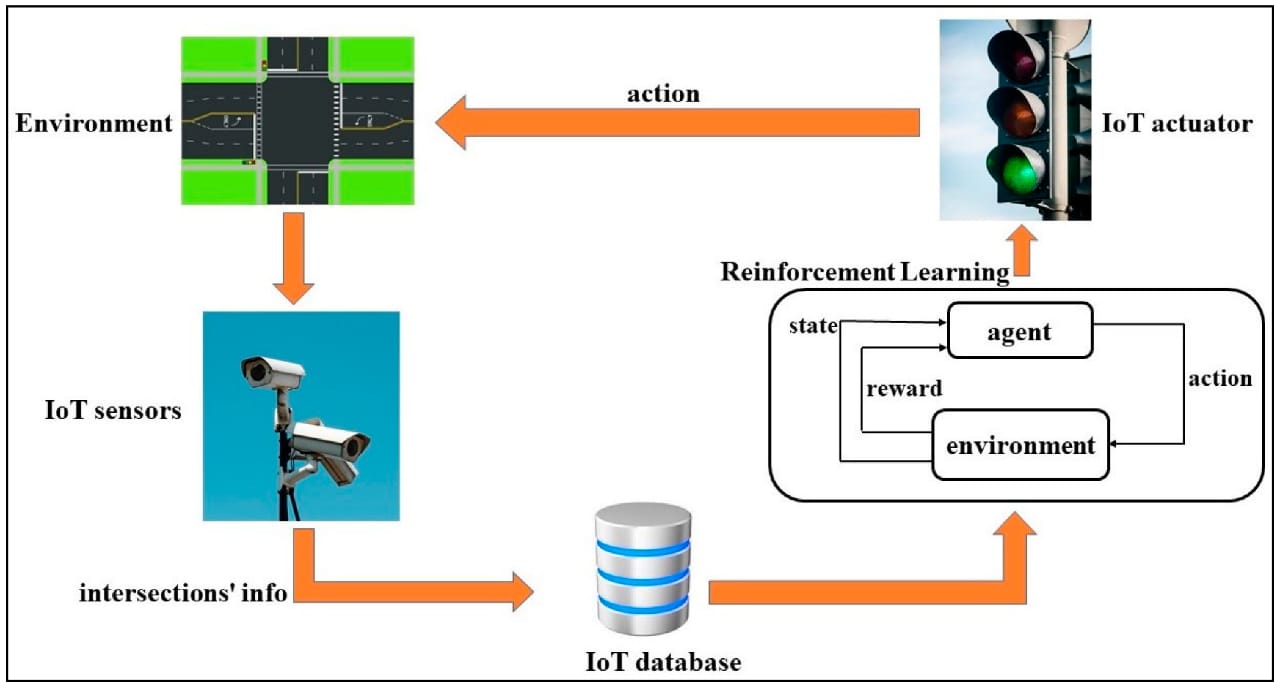


**Traffic Cameras**: High-resolution cameras for collecting visual data.

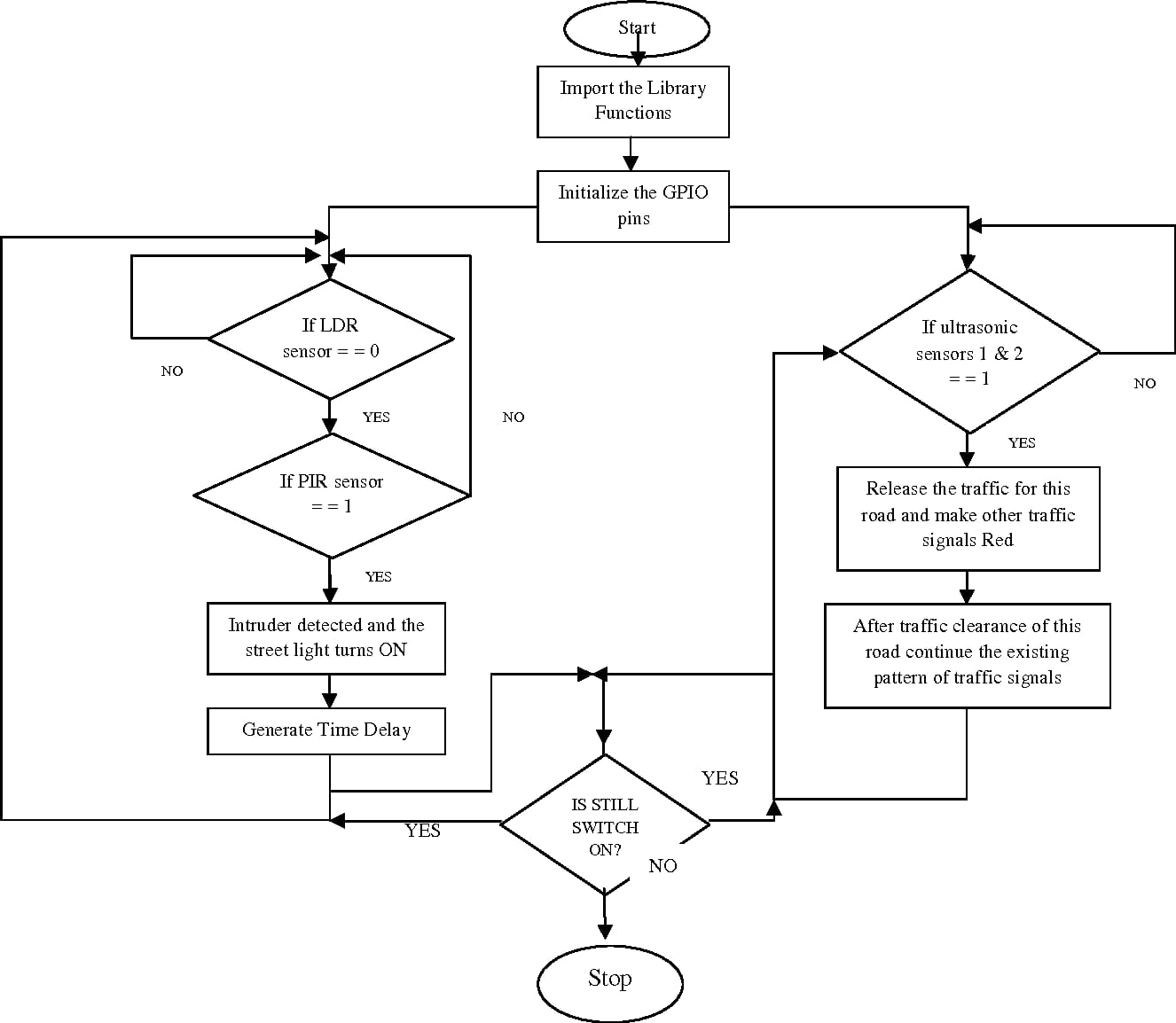
**PROPOSED SYSTEM ARCHITECTURE:**

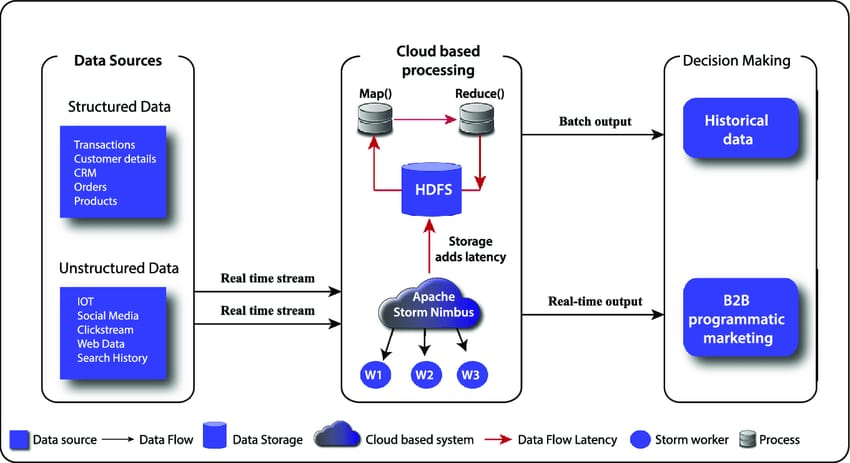


The system architecture of a Traffic Management System (TMS) outlines how various components, such as traffic sensors, data processing servers, machine learning models, and communication infrastructure, work together to monitor, analyze, and predict traffic conditions. It details how data flows from sensors to servers for real-time analysis, congestion prediction, and user interface display.

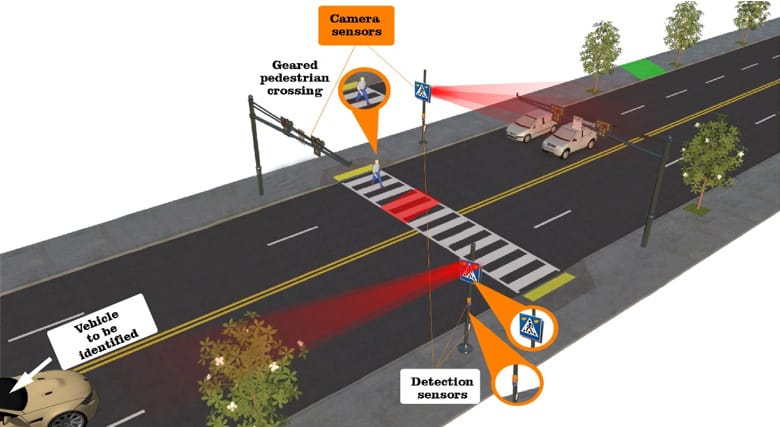


FLOW DIAGRAMS

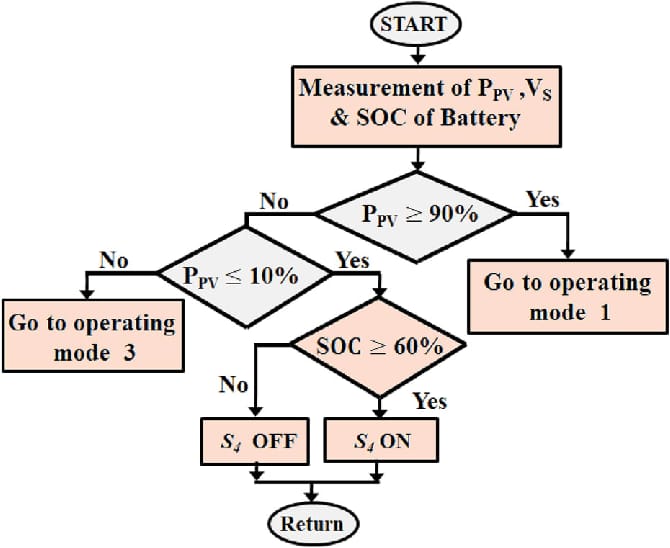


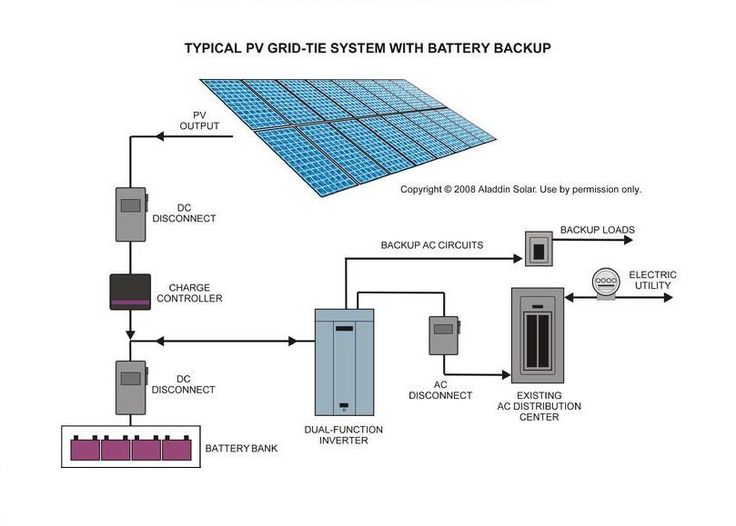
DATA PROCESSING DIAGRAM

CAMERA SETUP



POWER BACKUP SUSTEM FLOW CHART



PROGRAM

import cv2

import dlib

import time

import threading

import math

import helm

carCascade = cv2.CascadeClassifier('cars.xml')

bikeCascade = cv2.CascadeClassifier('motor-v4.xml')

video = cv2.VideoCapture('test.mp4')

LAG=7

WIDTH = 1280

HEIGHT = 720

OPTIMISE= 7

def estimateSpeed(location1, location2,fps):

d\_pixels = math.sqrt(math.pow(location2[0] - location1[0], 2) + math.pow(location2[1] - location1[1], 2))

# ppm = location2[2] / carWidht

ppm = 8.8

d\_meters = d\_pixels / ppm

if fps == 0.0:

fps = 18

speed = d\_meters \* fps \* 3.6

return speed

def trackMultipleObjects():

rectangleColor = (0, 255, 0)

frameCounter = 0

currentCarID = 0

currentBikeID=0

fps = 0

carTracker = {}

bikeTracker = {}

bikeNumbers = {}

carNumbers = {}

bikeLocation1 = {}

carLocation1 = {}

bikeLocation2 = {}

carLocation2 = {}

speed = [None] \* 1000

go =[False for i in range(1000)]

identity = [0 for i in range(1000)]

snaps = [False for i in range(1000)]

types = ["cars" for i in range(1000)]

Helmets = ["No Helmet Detected" for i in range(1000)]

out = cv2.VideoWriter('outpy.avi',cv2.VideoWriter\_fourcc('M','J','P','G'), 10, (WIDTH,HEIGHT))

while True:

start\_time = time.time()

rc, image = video.read()

if type(image) == type(None):

break

image = cv2.resize(image, (WIDTH, HEIGHT))

resultImage = image.copy()

frameCounter = frameCounter + 1

carIDtoDelete = []

for carID in carTracker.keys():

trackingQuality = carTracker[carID].update(image)

if trackingQuality < 7:

carIDtoDelete.append(carID)

for carID in carIDtoDelete:

print ('Removing carID ' + str(carID) + ' from list of trackers.')

print ('Removing carID ' + str(carID) + ' previous location.')

print ('Removing carID ' + str(carID) + ' current location.')

carTracker.pop(carID, None)

carLocation1.pop(carID, None)

carLocation2.pop(carID, None)

if not (frameCounter % 10):

gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

cars = carCascade.detectMultiScale(gray, 1.1, 13, 18, (24, 24))

bikes = bikeCascade.detectMultiScale(gray, 1.1 , 13, 18, (24,24))

for (\_x, \_y, \_w, \_h) in cars:

x = int(\_x)

y = int(\_y)

w = int(\_w)

h = int(\_h)

roi = image[y:y+h,x:x+w]

x\_bar = x + 0.5 \* w

y\_bar = y + 0.5 \* h

matchCarID = None

for carID in carTracker.keys():

trackedPosition = carTracker[carID].get\_position()

t\_x = int(trackedPosition.left())

t\_y = int(trackedPosition.top())

t\_w = int(trackedPosition.width())

t\_h = int(trackedPosition.height())

t\_x\_bar = t\_x + 0.5 \* t\_w

t\_y\_bar = t\_y + 0.5 \* t\_h

if ((t\_x <= x\_bar <= (t\_x + t\_w)) and (t\_y <= y\_bar <= (t\_y + t\_h)) and (x <= t\_x\_bar <= (x + w)) and (y <= t\_y\_bar <= (y + h))):

matchCarID = carID

if matchCarID is None:

print ('Creating new tracker ' + str(currentCarID))

tracker = dlib.correlation\_tracker()

tracker.start\_track(image, dlib.rectangle(x, y, x + w, y + h))

carTracker[currentCarID] = tracker

carLocation1[currentCarID] = [x, y, w, h]

currentCarID = currentCarID + 1

for (\_x, \_y, \_w, \_h) in bikes:

x = int(\_x)

y = int(\_y)

w = int(\_w)

h = int(\_h)

x\_bar = x + 0.5 \* w

y\_bar = y + 0.5 \* h

matchCarID = None

for carID in carTracker.keys():

trackedPosition = carTracker[carID].get\_position()

t\_x = int(trackedPosition.left())

t\_y = int(trackedPosition.top())

t\_w = int(trackedPosition.width())

t\_h = int(trackedPosition.height())

t\_x\_bar = t\_x + 0.5 \* t\_w

t\_y\_bar = t\_y + 0.5 \* t\_h

if ((t\_x <= x\_bar <= (t\_x + t\_w)) and (t\_y <= y\_bar <= (t\_y + t\_h)) and (x <= t\_x\_bar <= (x + w)) and (y <= t\_y\_bar <= (y + h))):

matchCarID = carID

if matchCarID is None:

print ('Creating new tracker ' + str(currentCarID))

tracker = dlib.correlation\_tracker()

tracker.start\_track(image, dlib.rectangle(x, y, x + w, y + h))

carTracker[currentCarID] = tracker

carLocation1[currentCarID] = [x, y, w, h]

types[currentCarID]= "bikes"

currentCarID = currentCarID + 1

for carID in carTracker.keys():

trackedPosition = carTracker[carID].get\_position()

t\_x = int(trackedPosition.left())

t\_y = int(trackedPosition.top())

t\_w = int(trackedPosition.width())

t\_h = int(trackedPosition.height())

cv2.rectangle(resultImage, (t\_x, t\_y), (t\_x + t\_w, t\_y + t\_h), rectangleColor, 4)

carLocation2[carID] = [t\_x, t\_y, t\_w, t\_h]

end\_time = time.time()

fps=0.0

for i in carLocation1.keys():

if frameCounter % 1 == 0:

[x1, y1, w1, h1] = carLocation1[i]

[x2, y2, w2, h2] = carLocation2[i]

carLocation1[i] = [x2, y2, w2, h2]

if [x1, y1, w1, h1] != [x2, y2, w2, h2]:

result = False

roi = resultImage[y1:y1+h1,x1:x1+w1]

if types[i]=="bikes" and Helmets[i] == "No Helmet Detected" and identity[i]< OPTIMISE:

result = helm.detect(roi)

if result==True:

Helmets[i]= "Helmet Detected"

if 7==7:

if not (end\_time == start\_time):

fps = 1.0/(end\_time - start\_time)

speed[i] = estimateSpeed([x1, y1, w1, h1], [x2, y2, w2, h2],fps)

if int(speed[i])>40:

speed[i]= speed[i]%40

if go[i] == True and int(speed[i])<10:

speed[i]=speed[i]+15

if int(speed[i])==0:

continue

if int(speed[i])>30:

go[i]=True

cv2.putText(resultImage, "OverSpeeding ALERT", (int(x1 + w1/2), int(y1-5)),cv2.FONT\_HERSHEY\_SIMPLEX, 0.75, (0, 0, 255), 2)

elif speed[i] != None and y1 >= 180 and speed[i]!=0:

ans= str(int(speed[i])) + " km/hr "

if types[i]=="bikes":

ans= ans+ Helmets[i]

cv2.putText(resultImage, ans, (int(x1 + w1/2), int(y1-5)),cv2.FONT\_HERSHEY\_SIMPLEX, 0.75, (0, 255, 0), 2)

identity[i]+=1

cv2.imshow('result', resultImage)

if cv2.waitKey(33) == 27:

break

cv2.destroyAllWindows()

if \_name\_ == '\_main\_':

trackMultipleObjects()

The main file to run the project…

from time import sleep

import cv2 as cv

import argparse

import sys

import numpy as np

import os.path

from glob import glob

#from PIL import image

frame\_count = 0 # used in mainloop where we're extracting images., and then to drawPred( called by post process)

frame\_count\_out=0 # used in post process loop, to get the no of specified class value.

# Initialize the parameters

confThreshold = 0.5 #Confidence threshold

nmsThreshold = 0.4 #Non-maximum suppression threshold

inpWidth = 416 #Width of network's input image

inpHeight = 416 #Height of network's input image

# Load names of classes

classesFile = "obj.names";

classes = None

with open(classesFile, 'rt') as f:

classes = f.read().rstrip('\n').split('\n')

# Give the configuration and weight files for the model and load the network using them.

modelConfiguration = "yolov3-obj.cfg";

modelWeights = "yolov3-obj\_2400.weights";

net = cv.dnn.readNetFromDarknet(modelConfiguration, modelWeights)

net.setPreferableBackend(cv.dnn.DNN\_BACKEND\_OPENCV)

net.setPreferableTarget(cv.dnn.DNN\_TARGET\_CPU)

# Get the names of the output layers

def getOutputsNames(net):

# Get the names of all the layers in the network

layersNames = net.getLayerNames()

# Get the names of the output layers, i.e. the layers with unconnected outputs

return [layersNames[i-1] for i in net.getUnconnectedOutLayers()]

# Draw the predicted bounding box

def drawPred(classId, conf, left, top, right, bottom, frame):

global frame\_count

# Draw a bounding box.

#cv.rectangle(frame, (left, top), (right, bottom), (255, 178, 50), 3)

label = '%.2f' % conf

# Get the label for the class name and its confidence

if classes:

assert(classId < len(classes))

label = '%s:%s' % (classes[classId], label)

#Display the label at the top of the bounding box

labelSize, baseLine = cv.getTextSize(label, cv.FONT\_HERSHEY\_SIMPLEX, 0.5, 1)

top = max(top, labelSize[1])

#print(label) #testing

#print(labelSize) #testing

#print(baseLine) #testing

label\_name,label\_conf = label.split(':') #spliting into class & confidance. will compare it with person.

if label\_name == 'Helmet':

#will try to print of label have people.. or can put a counter to find the no of people occurance.

#will try if it satisfy the condition otherwise, we won't print the boxes or leave it.

#cv.rectangle(frame, (left, top - round(1.5\*labelSize[1])), (left + round(1.5\*labelSize[0]), top + baseLine), (255, 255, 255), cv.FILLED)

#cv.putText(frame, label, (left, top), cv.FONT\_HERSHEY\_SIMPLEX, 0.75, (0,0,0), 1)

frame\_count+=1

#print(frame\_count)

if(frame\_count> 0):

return frame\_count

# Remove the bounding boxes with low confidence using non-maxima suppression

def postprocess(frame, outs):

frameHeight = frame.shape[0]

frameWidth = frame.shape[1]

frame\_count\_out=0

classIds = []

confidences = []

boxes = []

# Scan through all the bounding boxes output from the network and keep only the

# ones with high confidence scores. Assign the box's class label as the class with the highest score.

classIds = [] #have to fins which class have hieghest confidence........=====>>><<<<=======

confidences = []

boxes = []

for out in outs:

for detection in out:

scores = detection[5:]

classId = np.argmax(scores)

confidence = scores[classId]

if confidence > confThreshold:

center\_x = int(detection[0] \* frameWidth)

center\_y = int(detection[1] \* frameHeight)

width = int(detection[2] \* frameWidth)

height = int(detection[3] \* frameHeight)

left = int(center\_x - width / 2)

top = int(center\_y - height / 2)

classIds.append(classId)

#print(classIds)

confidences.append(float(confidence))

boxes.append([left, top, width, height])

# Perform non maximum suppression to eliminate redundant overlapping boxes with

# lower confidences.

indices = cv.dnn.NMSBoxes(boxes, confidences, confThreshold, nmsThreshold)

count\_person=0 # for counting the classes in this loop.

for i in indices:

i = i[0]

box = boxes[i]

left = box[0]

top = box[1]

width = box[2]

height = box[3]

#this function in loop is calling drawPred so, try pushing one test counter in parameter , so it can calculate it.

frame\_count\_out = drawPred(classIds[i], confidences[i], left, top, left + width, top + height, frame)

#increase test counter till the loop end then print...

#checking class, if it is a person or not

my\_class='Helmet' #======================================== mycode .....

unknown\_class = classes[classId]

if my\_class == unknown\_class:

count\_person += 1

#if(frame\_count\_out > 0):

#print(frame\_count\_out)

if count\_person >= 1:

path = 'test\_out/'

# frame\_name=os.path.basename(fn) # trimm the path and give file name.

#cv.imwrite(str(path)+frame\_name, frame) # writing to folder.

#print(type(frame))

#cv.imshow('img',frame)

#cv.waitKey(800)

return 1

else:

return 0

#cv.imwrite(frame\_name, frame)

# Process inputs

winName = 'Deep learning object detection in OpenCV'

cv.namedWindow(winName, cv.WINDOW\_NORMAL)

def detect(frame):

#frame = cv.imread(fn)

frame\_count =0

# Create a 4D blob from a frame.

blob = cv.dnn.blobFromImage(frame, 1/255, (inpWidth, inpHeight), [0,0,0], 1, crop=False)

# Sets the input to the network

net.setInput(blob)

# Runs the forward pass to get output of the output layers

outs = net.forward(getOutputsNames(net))

# Remove the bounding boxes with low confidence

# Put efficiency information. The function getPerfProfile returns the overall time for inference(t) and the timings for each of the layers(in layersTimes)

t, \_ = net.getPerfProfile()

#print(t)

label = 'Inference time: %.2f ms' % (t \* 1000.0 / cv.getTickFrequency())

#print(label)

#cv.putText(frame, label, (0, 15), cv.FONT\_HERSHEY\_SIMPLEX, 0.5, (0, 0, 255))

#print(label)

k=postprocess(frame, outs)

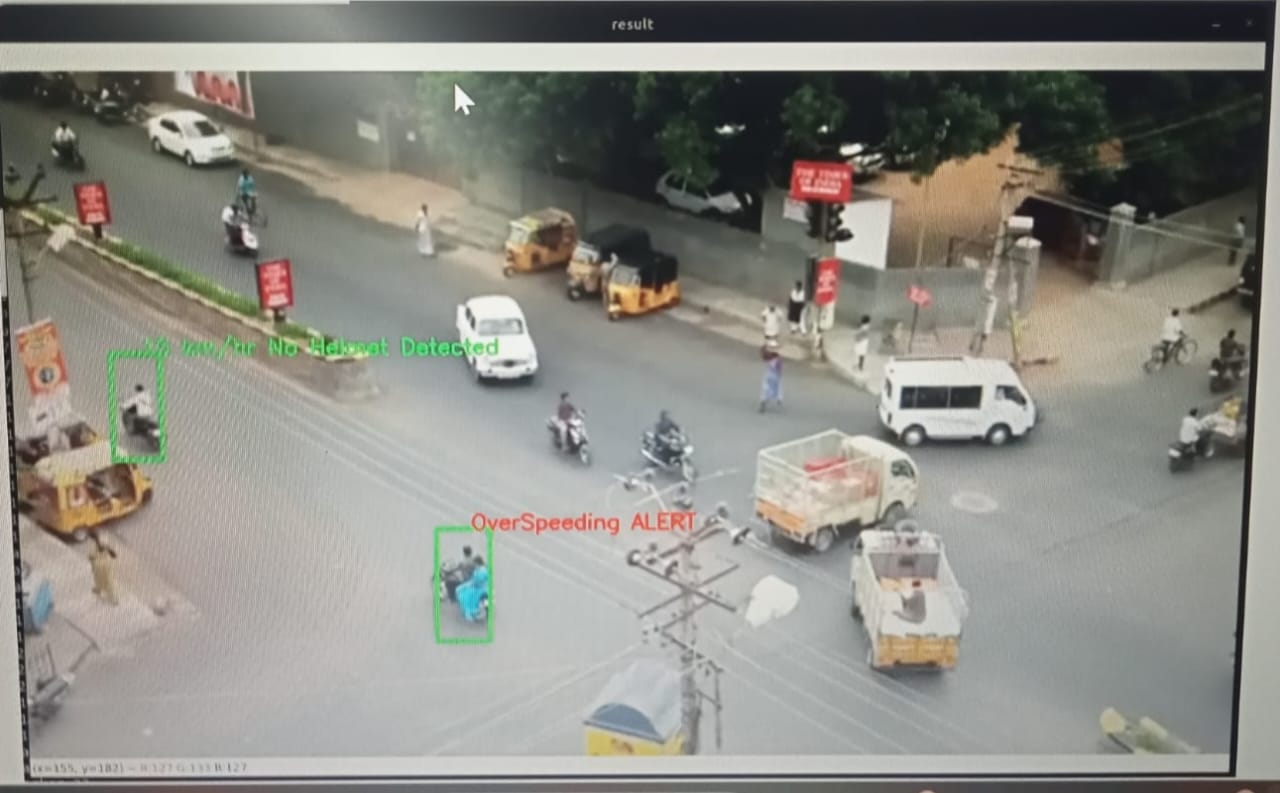
if k:

return 1

else:

        return 0

**OUTPUT**



**CONCLUSION:**

**Project Overview:** The Traffic Management System (TMS) is a comprehensive project aimed at efficiently managing and controlling traffic flow on roadways and in urban areas. It integrates historical traffic data, realtime data collection through IoT, and machine learning algorithms to predict congestion patterns and enhance overall traffic management.

**Project Output:** The primary output of the project is an intelligent traffic management system capable of predicting traffic congestion in real-time. It provides insights into traffic conditions, allows for proactive decision-making by authorities, and communicates this information to the public.

**Result and Functionality:** The result of the project is an advanced TMS that continuously collects, processes, and analyzes traffic data. Machine learning models predict congestion patterns and provide timely alerts. The system's effectiveness is based on historical data and real-time inputs, enabling authorities to take proactive measures to alleviate congestion and improve overall traffic flow.

**Public Benefits:** The TMS project greatly benefits the public by reducing congestion, minimizing travel time, enhancing road safety, and promoting eco-friendly transportation practices. Real-time traffic information is disseminated through various channels, allowing commuters to make informed decisions about their routes. This not only reduces stress and frustration but also contributes to environmental sustainability through reduced fuel consumption and emissions. Ultimately, the Traffic Management System serves the public by making their daily commutes smoother and more efficient.