**COMPUTER VISION PROJECT**



***People in and out counter using OpenCV***

**SUBMITTED TO:**

**Dr. RAJIV KUMAR KAPOOR**

**SUBMITTED BY:**

**Hitik Kumar**

**2K18/MC/042**

**PROJECT DESCRIPTION**

The main idea behind the project is to count number of people going in and out of a certain place. Application of this could be in some place like a super market , where for security reasons you want to check how many people currently inside the market and also assure that everyone went outside. This is also beneficial for using it for a statistical study, about how many people visit your store on weekdays and weekends and maybe how different it is during holidays, to accurately predict your profit and spend your money in accordance.

The next step is to include facial recognition software in these with a better angle view and clearer videos. The application then increases to utilization in schools for attendance or checking if a student is bunking etc. and even in offices and other areas. We’ll use OpenCV for standard computer vision/image processing functions, along with the deep learning object detector for people counting.

We’ll then use dlib for its implementation of correlation filters. We could use OpenCV here as well; however, the dlib object tracking implementation was a bit easier to work with for this project. The main idea is to use both an object detector and an object tracker to optimize results.

**Code:**

Main-

from imutils.video import VideoStream

from imutils.video import FPS

import numpy as np

import argparse

import imutils

import time

import dlib

import cv2

from pyimagesearch.centroidtracker import CentroidTracker

from pyimagesearch.trackableobject import TrackableObject

ap = argparse.ArgumentParser()

ap.add\_argument("-p", "--prototxt", required=True,

help="path to Caffe 'deploy' prototxt file")

ap.add\_argument("-m", "--model", required=True,

help="path to Caffe pre-trained model")

ap.add\_argument("-i", "--input", type=str,

help="path to optional input video file")

ap.add\_argument("-o", "--output", type=str,

help="path to optional output video file")

ap.add\_argument("-c", "--confidence", type=float, default=0.4,

help="minimum probability to filter weak detections")

ap.add\_argument("-s", "--skip-frames", type=int, default=30,

help="# of skip frames between detections")

args = vars(ap.parse\_args())

# initialize the list of class labels MobileNet SSD was trained to

# detect

CLASSES = ["background", "aeroplane", "bicycle", "bird", "boat",

"bottle", "bus", "car", "cat", "chair", "cow", "diningtable",

"dog", "horse", "motorbike", "person", "pottedplant", "sheep",

"sofa", "train", "tvmonitor"]

print("[INFO] loading model...")

net = cv2.dnn.readNetFromCaffe(args["prototxt"], args["model"])

if not args.get("input", False):

print("[INFO] starting video stream...")

vs = VideoStream(src=0).start()

time.sleep(2.0)

else:

print("[INFO] opening video file...")

vs = cv2.VideoCapture(args["input"])

writer = None

W = None

H = None

ct = CentroidTracker(maxDisappeared=40, maxDistance=50)

trackers = []

trackableObjects = {}

totalFrames = 0

totalDown = 0

totalUp = 0

fps = FPS().start()

while True:

frame = vs.read()

frame = frame[1] if args.get("input", False) else frame

if args["input"] is not None and frame is None:

break

frame = imutils.resize(frame, width=500)

rgb = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)

if W is None or H is None:

(H, W) = frame.shape[:2]

if args["output"] is not None and writer is None:

fourcc = cv2.VideoWriter\_fourcc(\*"MJPG")

writer = cv2.VideoWriter(args["output"], fourcc, 30,

(W, H), True)

status = "Waiting"

rects = []

if totalFrames % args["skip\_frames"] == 0:

status = "Detecting"

trackers = []

blob = cv2.dnn.blobFromImage(frame, 0.007843, (W, H), 127.5)

net.setInput(blob)

detections = net.forward()

for i in np.arange(0, detections.shape[2]):

confidence = detections[0, 0, i, 2]

if confidence > args["confidence"]:

idx = int(detections[0, 0, i, 1])

if CLASSES[idx] != "person":

continue

box = detections[0, 0, i, 3:7] \* np.array([W, H, W, H])

(startX, startY, endX, endY) = box.astype("int")

tracker = dlib.correlation\_tracker()

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

tracker.start\_track(rgb, rect)

trackers.append(tracker)

else:

for tracker in trackers:

status = "Tracking"

tracker.update(rgb)

pos = tracker.get\_position()

startX = int(pos.left())

startY = int(pos.top())

endX = int(pos.right())

endY = int(pos.bottom())

rects.append((startX, startY, endX, endY))

cv2.line(frame, (0, H // 2), (W, H // 2), (0, 255, 255), 2)

objects = ct.update(rects)

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

to = trackableObjects.get(objectID, None)

if to is None:

to = TrackableObject(objectID, centroid)

else:

y = [c[1] for c in to.centroids]

direction = centroid[1] - np.mean(y)

to.centroids.append(centroid)

if not to.counted:

if direction < 0 and centroid[1] < H // 2:

totalUp += 1

to.counted = True

elif direction > 0 and centroid[1] > H // 2:

totalDown += 1

to.counted = True

trackableObjects[objectID] = to

text = "ID {}".format(objectID)

cv2.putText(frame, text, (centroid[0] - 10, centroid[1] - 10),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (0, 255, 0), 2)

cv2.circle(frame, (centroid[0], centroid[1]), 4, (0, 255, 0), -1)

info = [

("Up", totalUp),

("Down", totalDown),

("Status", status),

]

for (i, (k, v)) in enumerate(info):

text = "{}: {}".format(k, v)

cv2.putText(frame, text, (10, H - ((i \* 20) + 20)),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.6, (0, 0, 255), 2)

if writer is not None:

writer.write(frame)

cv2.imshow("Frame", frame)

key = cv2.waitKey(1) & 0xFF

if key == ord("q"):

break

totalFrames += 1

fps.update()

fps.stop()

print("[INFO] elapsed time: {:.2f}".format(fps.elapsed()))

print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))

if writer is not None:

writer.release()

if not args.get("input", False):

vs.stop()

else:

vs.release()

cv2.destroyAllWindows()

#Centroid tracker

# import the necessary packages

from scipy.spatial import distance as dist

from collections import OrderedDict

import numpy as np

class CentroidTracker:

def \_\_init\_\_(self, maxDisappeared=50, maxDistance=50):

# initialize the next unique object ID along with two ordered

# dictionaries used to keep track of mapping a given object

# ID to its centroid and number of consecutive frames it has

# been marked as "disappeared", respectively

self.nextObjectID = 0

self.objects = OrderedDict()

self.disappeared = OrderedDict()

# store the number of maximum consecutive frames a given

# object is allowed to be marked as "disappeared" until we

# need to deregister the object from tracking

self.maxDisappeared = maxDisappeared

# store the maximum distance between centroids to associate

# an object -- if the distance is larger than this maximum

# distance we'll start to mark the object as "disappeared"

self.maxDistance = maxDistance

def register(self, centroid):

# when registering an object we use the next available object

# ID to store the centroid

self.objects[self.nextObjectID] = centroid

self.disappeared[self.nextObjectID] = 0

self.nextObjectID += 1

def deregister(self, objectID):

# to deregister an object ID we delete the object ID from

# both of our respective dictionaries

del self.objects[objectID]

del self.disappeared[objectID]

def update(self, rects):

# check to see if the list of input bounding box rectangles

# is empty

if len(rects) == 0:

# loop over any existing tracked objects and mark them

# as disappeared

for objectID in list(self.disappeared.keys()):

self.disappeared[objectID] += 1

# if we have reached a maximum number of consecutive

# frames where a given object has been marked as

# missing, deregister it

if self.disappeared[objectID] > self.maxDisappeared:

self.deregister(objectID)

# return early as there are no centroids or tracking info

# to update

return self.objects

# initialize an array of input centroids for the current frame

inputCentroids = np.zeros((len(rects), 2), dtype="int")

# loop over the bounding box rectangles

for (i, (startX, startY, endX, endY)) in enumerate(rects):

# use the bounding box coordinates to derive the centroid

cX = int((startX + endX) / 2.0)

cY = int((startY + endY) / 2.0)

inputCentroids[i] = (cX, cY)

# if we are currently not tracking any objects take the input

# centroids and register each of them

if len(self.objects) == 0:

for i in range(0, len(inputCentroids)):

self.register(inputCentroids[i])

# otherwise, are are currently tracking objects so we need to

# try to match the input centroids to existing object

# centroids

else:

# grab the set of object IDs and corresponding centroids

objectIDs = list(self.objects.keys())

objectCentroids = list(self.objects.values())

# compute the distance between each pair of object

# centroids and input centroids, respectively -- our

# goal will be to match an input centroid to an existing

# object centroid

D = dist.cdist(np.array(objectCentroids), inputCentroids)

# in order to perform this matching we must (1) find the

# smallest value in each row and then (2) sort the row

# indexes based on their minimum values so that the row

# with the smallest value as at the \*front\* of the index

# list

rows = D.min(axis=1).argsort()

# next, we perform a similar process on the columns by

# finding the smallest value in each column and then

# sorting using the previously computed row index list

cols = D.argmin(axis=1)[rows]

# in order to determine if we need to update, register,

# or deregister an object we need to keep track of which

# of the rows and column indexes we have already examined

usedRows = set()

usedCols = set()

# loop over the combination of the (row, column) index

# tuples

for (row, col) in zip(rows, cols):

# if we have already examined either the row or

# column value before, ignore it

if row in usedRows or col in usedCols:

continue

# if the distance between centroids is greater than

# the maximum distance, do not associate the two

# centroids to the same object

if D[row, col] > self.maxDistance:

continue

# otherwise, grab the object ID for the current row,

# set its new centroid, and reset the disappeared

# counter

objectID = objectIDs[row]

self.objects[objectID] = inputCentroids[col]

self.disappeared[objectID] = 0

# indicate that we have examined each of the row and

# column indexes, respectively

usedRows.add(row)

usedCols.add(col)

# compute both the row and column index we have NOT yet

# examined

unusedRows = set(range(0, D.shape[0])).difference(usedRows)

unusedCols = set(range(0, D.shape[1])).difference(usedCols)

# in the event that the number of object centroids is

# equal or greater than the number of input centroids

# we need to check and see if some of these objects have

# potentially disappeared

if D.shape[0] >= D.shape[1]:

# loop over the unused row indexes

for row in unusedRows:

# grab the object ID for the corresponding row

# index and increment the disappeared counter

objectID = objectIDs[row]

self.disappeared[objectID] += 1

# check to see if the number of consecutive

# frames the object has been marked "disappeared"

# for warrants deregistering the object

if self.disappeared[objectID] > self.maxDisappeared:

self.deregister(objectID)

# otherwise, if the number of input centroids is greater

# than the number of existing object centroids we need to

# register each new input centroid as a trackable object

else:

for col in unusedCols:

self.register(inputCentroids[col])

# return the set of trackable objects

return self.objects

#trackable object –

class TrackableObject:

def \_\_init\_\_(self, objectID, centroid):

# store the object ID, then initialize a list of centroids

# using the current centroid

self.objectID = objectID

self.centroids = [centroid]

# initialize a boolean used to indicate if the object has

# already been counted or not

self.counted = False

**Results:**

Snippets of results-





