```python
# USAGE
# To read and write back out to video:
# python people_counter.py --prototxt mobilenet_ssd/MobileNetSSD_deploy.prototxt \
#       --model mobilenet_ssd/MobileNetSSD_deploy.caffemodel --input
videos/example_01.mp4 \
#       --output output/output_01.avi
#
# To read from webcam and write back out to disk:
# python people_counter.py --prototxt mobilenet_ssd/MobileNetSSD_deploy.prototxt \
#       --model mobilenet_ssd/MobileNetSSD_deploy.caffemodel \
#       --output output/webcam_output.avi

# import the necessary packages
from pyimagesearch.centroidtracker import CentroidTracker
from pyimagesearch.trackableobject import TrackableObject
from imutils.video import VideoStream
from imutils.video import FPS
import numpy as np
import argparse
import imutils
import time
import dlib
import cv2

# construct the argument parse and parse the arguments
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"]

# load our serialized model from disk
print("[INFO] loading model...")
net = cv2.dnn.readNetFromCaffe(args["prototxt"], args["model"])

# if a video path was not supplied, grab a reference to the webcam
if not args.get("input", False):
        print("[INFO] starting video stream...")
        vs = VideoStream(src=0).start()
        time.sleep(2.0)

# otherwise, grab a reference to the video file
```

```
else:
        print("[INFO] opening video file...")
        vs = cv2.VideoCapture(args["input"])

# initialize the video writer (we'll instantiate later if need be)
writer = None

# initialize the frame dimensions (we'll set them as soon as we read
# the first frame from the video)
W = None
H = None

# instantiate our centroid tracker, then initialize a list to store
# each of our dlib correlation trackers, followed by a dictionary to
# map each unique object ID to a TrackableObject
ct = CentroidTracker(maxDisappeared=40, maxDistance=50)
trackers = []
trackableObjects = {}

# initialize the total number of frames processed thus far, along
# with the total number of objects that have moved either up or down
totalFrames = 0
totalDown = 0
totalUp = 0

# start the frames per second throughput estimator
fps = FPS().start()

# loop over frames from the video stream
while True:
        # grab the next frame and handle if we are reading from either
        # VideoCapture or VideoStream
        frame = vs.read()
        frame = frame[1] if args.get("input", False) else frame

        # if we are viewing a video and we did not grab a frame then we
        # have reached the end of the video
        if args["input"] is not None and frame is None:
                break

        # resize the frame to have a maximum width of 500 pixels (the
        # less data we have, the faster we can process it), then convert
        # the frame from BGR to RGB for dlib
        frame = imutils.resize(frame, width=500)
        rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

        # if the frame dimensions are empty, set them
        if W is None or H is None:
                (H, W) = frame.shape[:2]

        # if we are supposed to be writing a video to disk, initialize
        # the writer
        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)

        # initialize the current status along with our list of bounding
        # box rectangles returned by either (1) our object detector or
```

```
        # (2) the correlation trackers
        status = "Waiting"
        rects = []

        # check to see if we should run a more computationally expensive
        # object detection method to aid our tracker
        if totalFrames % args["skip_frames"] == 0:
                # set the status and initialize our new set of object trackers
                status = "Detecting"
                trackers = []

                # convert the frame to a blob and pass the blob through the
                # network and obtain the detections
                blob = cv2.dnn.blobFromImage(frame, 0.007843, (W, H), 127.5)
                net.setInput(blob)
                detections = net.forward()

                # loop over the detections
                for i in np.arange(0, detections.shape[2]):
                        # extract the confidence (i.e., probability) associated
                        # with the prediction
                        confidence = detections[0, 0, i, 2]

                        # filter out weak detections by requiring a minimum
                        # confidence
                        if confidence > args["confidence"]:
                                # extract the index of the class label from the
                                # detections list
                                idx = int(detections[0, 0, i, 1])

                                # if the class label is not a person, ignore it
                                if CLASSES[idx] != "person":
                                        continue

                                # compute the (x, y)-coordinates of the bounding box
                                # for the object
                                box = detections[0, 0, i, 3:7] * np.array([W, H, W, H])
                                (startX, startY, endX, endY) = box.astype("int")

                                # construct a dlib rectangle object from the bounding
                                # box coordinates and then start the dlib correlation
                                # tracker
                                tracker = dlib.correlation_tracker()
                                rect = dlib.rectangle(startX, startY, endX, endY)
                                tracker.start_track(rgb, rect)

                                # add the tracker to our list of trackers so we can
                                # utilize it during skip frames
                                trackers.append(tracker)

        # otherwise, we should utilize our object *trackers* rather than
        # object *detectors* to obtain a higher frame processing throughput
        else:
                # loop over the trackers
                for tracker in trackers:
                        # set the status of our system to be 'tracking' rather
                        # than 'waiting' or 'detecting'
                        status = "Tracking"

```

```
                        # update the tracker and grab the updated position
                        tracker.update(rgb)
                        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 rectangles list
                        rects.append((startX, startY, endX, endY))

        # draw a horizontal line in the center of the frame -- once an
        # object crosses this line we will determine whether they were
        # moving 'up' or 'down'
        cv2.line(frame, (0, H // 2), (W, H // 2), (0, 255, 255), 2)

        # use the centroid tracker to associate the (1) old object
        # centroids with (2) the newly computed object centroids
        objects = ct.update(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 there is no existing trackable object, create one
                if to is None:
                        to = TrackableObject(objectID, centroid)

                # otherwise, there is a trackable object so we can utilize it
                # to determine direction
                else:
                        # the difference between the y-coordinate of the *current*
                        # centroid and the mean of *previous* centroids will tell
                        # us in which direction the object is moving (negative for
                        # 'up' and positive for 'down')
                        y = [c[1] for c in to.centroids]
                        direction = centroid[1] - np.mean(y)
                        to.centroids.append(centroid)

                        # check to see if the object has been counted or not
                        if not to.counted:
                                # if the direction is negative (indicating the object
                                # is moving up) AND the centroid is above the center
                                # line, count the object
                                if direction < 0 and centroid[1] < H // 2:
                                        totalUp += 1
                                        to.counted = True

                                # if the direction is positive (indicating the object
                                # is moving down) AND the centroid is below the
                                # center line, count the object
                                elif direction > 0 and centroid[1] > H // 2:
                                        totalDown += 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
                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)

        # construct a tuple of information we will be displaying on the
        # frame
        info = [
                ("Up", totalUp),
                ("Down", totalDown),
                ("Status", status),
        ]

        # loop over the info tuples and draw them on our frame
        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)

        # check to see if we should write the frame to disk
        if writer is not None:
                writer.write(frame)

        # show the output frame
        cv2.imshow("Frame", frame)
        key = cv2.waitKey(1) & 0xFF

        # if the `q` key was pressed, break from the loop
        if key == ord("q"):
                break

        # increment the total number of frames processed thus far and
        # then update the FPS counter
        totalFrames += 1
        fps.update()

# stop the timer and display FPS information
fps.stop()
print("[INFO] elapsed time: {:.2f}".format(fps.elapsed()))
print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))

# check to see if we need to release the video writer pointer
if writer is not None:
        writer.release()

# if we are not using a video file, stop the camera video stream
if not args.get("input", False):
        vs.stop()

# otherwise, release the video file pointer
else:
        vs.release()

# close any open windows
```

```
294 cv2.destroyAllWindows()
```