**Video analytics and Motion detection**

Video analytics use artificial intelligence to complete various tasks by applying computer vision and deep learning to video footage or live video streams. Video analytics is also sometimes referred to as video content analysis or intelligent video analytics.

**Importance**

Novel video analytics solutions are quickly gaining traction. Key adopters are both companies seeking to use the latest AI technologies to solve long-standing problems and those that have been operating video surveillance systems before the emergence of Artificial intelligence (AI). Advances made in Deep and Machine Learning, both subsets of AI, have enabled video analytics to revolutionise the task automation landscape, allowing tasks that previously required human intervention to be successfully automated to be automated. Video Intelligence has pre-trained machine learning models that automatically recognize a vast number of objects, places, and actions in stored and streaming video. Offering exceptional quality out of the box, it’s highly efficient for common use cases and improves over time as new concepts are introduced. The market for video analytics is always changing. The most recent advances in video analytics use Deep Learning for video content analysis, the ability to conduct real-time video processing, and the heightened accuracy of video recognition software.

**Video Analytics in Specified Industries**

**Vertical Motion Detection**

A specific instance of video analytics for security could be a fence-climbing detection system. Security staff is usually trained to know that people walking outside a fence is considered regular, but climbing on top of or struggling with the fence is irregular. Video analytics software trained to recognize the subtle differences in motion direction between the regular and irregular behavior involving the fence can be linked to the real-time video feed from security cameras.

**Loitering Detection**

In Smart Cities, video analytics are trained to notice when people or vehicles remain in a defined zone longer than the user-defined time allows. For the safety of the area, an alarm could be activated depending on the preferences of the program implementer. This behavior is effective in the real-time notification of suspicious behavior around pharmacy departments, ATMs, narcotic dispensaries, and other locations.

**People counting**

People counting can be conducted using video analytics. Retail involves a lot of experimenting with displays and marketing strategies. Observing or having access to how many customers come in and when is helpful for stores to know what is working in terms of marketing and product overview. In addition, noticing how many customers spend prolonged periods of time near which displays are useful for the store because it improves the customer experience and business for the store. In terms of people counting, video analytics provides operational insights and branding insights and reveals a host of other aspects of customer relationships.

**Home monitoring**

Home monitoring of older adults or people with health issues is another example of an application that provides great value. For instance, falls are a major cause of injury and death in older persons. Although personal medical devices can detect falls, they must be worn and are frequently disregarded by the consumer. A video analytics solution can process the signals of home cameras to detect in real time if a person has fallen. With proper setup, such a system could also determine if a person took a given medication when they were supposed to, for instance.

**Example**

The colab notebook is counts road traffic using Python and OpenCV with the idea of motion detection as well as background subtraction algorithm.

CELL1:

# install library and download video for processing

!pip install sk-video>=1.1.8

import os

if not os.path.exists('road.mp4'):

    !wget https://learnml.s3.eu-north-1.amazonaws.com/road.mp4

CELL2:

# import needed modules

import os

import csv

import numpy as np

import logging

import logging.handlers

import math

import sys

import random

import numpy as np

import skvideo.io

import cv2

import matplotlib.pyplot as plt

from IPython.display import HTML

from base64 import b64encode

# without this some strange errors happen

cv2.ocl.setUseOpenCL(False)

random.seed(123)

# setup logging

def init\_logging(level=logging.INFO):

    main\_logger = logging.getLogger()

    for hnd in main\_logger.handlers:

        main\_logger.removeHandler(hnd)

    formatter = logging.Formatter(

        fmt='%(asctime)s.%(msecs)03d %(levelname)-8s [%(name)s] %(message)s', datefmt='%Y-%m-%d %H:%M:%S')

    handler\_stream = logging.StreamHandler(sys.stdout)

    handler\_stream.setFormatter(formatter)

    main\_logger.addHandler(handler\_stream)

    main\_logger.setLevel(level)

    return main\_logger

CELL3:

def train\_bg\_subtractor(inst, cap, num=500):

    print ('Training BG Subtractor...')

    i = 0

    for frame in cap:

        inst.apply(frame, None, 0.001)

        i += 1

        if i >= num:

            return cap

VIDEO\_SOURCE = "road.mp4"

bg\_subtractor = cv2.createBackgroundSubtractorMOG2(

        history=500, detectShadows=True)

# Set up image source

cap = skvideo.io.vreader(VIDEO\_SOURCE)

# skipping 500 frames to train bg subtractor

train\_bg\_subtractor(bg\_subtractor, cap, num=500)

frame = next(cap)

fg\_mask = bg\_subtractor.apply(frame, None, 0.001)

plt.figure(figsize=(12,12))

plt.imshow(fg\_mask)

plt.show()

CELL4:

def filter\_mask(img):

    kernel = cv2.getStructuringElement(cv2.MORPH\_ELLIPSE, (2, 2))

    # Fill any small holes

    closing = cv2.morphologyEx(img, cv2.MORPH\_CLOSE, kernel)

    # Remove noise

    opening = cv2.morphologyEx(closing, cv2.MORPH\_OPEN, kernel)

    # Dilate to merge adjacent blobs

    dilation = cv2.dilate(opening, kernel, iterations=2)

    return dilation

bg\_subtractor = cv2.createBackgroundSubtractorMOG2(

        history=500, detectShadows=True)

# Set up image source

cap = skvideo.io.vreader(VIDEO\_SOURCE)

# skipping 500 frames to train bg subtractor

train\_bg\_subtractor(bg\_subtractor, cap, num=500)

frame = next(cap)

fg\_mask = bg\_subtractor.apply(frame, None, 0.001)

fg\_mask[fg\_mask < 240] = 0

fg\_mask = filter\_mask(fg\_mask)

plt.figure(figsize=(12,12))

plt.imshow(fg\_mask)

plt.show()

CELL5:

def get\_centroid(x, y, w, h):

    x1 = int(w / 2)

    y1 = int(h / 2)

    cx = x + x1

    cy = y + y1

    return (cx, cy)

class ContourDetection:

    '''  Detecting moving objects.

    '''

    def \_\_init\_\_(self, bg\_subtractor, min\_contour\_width=35, min\_contour\_height=35, save\_image=False, image\_dir='images'):

        super(ContourDetection, self).\_\_init\_\_()

        self.bg\_subtractor = bg\_subtractor

        self.min\_contour\_width = min\_contour\_width

        self.min\_contour\_height = min\_contour\_height

        self.save\_image = save\_image

        self.image\_dir = image\_dir

    def filter\_mask(self, img, a=None):

        '''

            This filters are hand-picked just based on visual tests

        '''

        kernel = cv2.getStructuringElement(cv2.MORPH\_ELLIPSE, (2, 2))

        # Fill any small holes

        closing = cv2.morphologyEx(img, cv2.MORPH\_CLOSE, kernel)

        # Remove noise

        opening = cv2.morphologyEx(closing, cv2.MORPH\_OPEN, kernel)

        # Dilate to merge adjacent blobs

        dilation = cv2.dilate(opening, kernel, iterations=2)

        return dilation

    def detect\_vehicles(self, fg\_mask):

        matches = []

        # finding external contours

        contours, hierarchy = cv2.findContours(

            fg\_mask, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_TC89\_L1)

        for (i, contour) in enumerate(contours):

            (x, y, w, h) = cv2.boundingRect(contour)

            # On the exit, we add some filtering by height, width and add centroid.

            contour\_valid = (w >= self.min\_contour\_width) and (

                h >= self.min\_contour\_height)

            if not contour\_valid:

                continue

            centroid = get\_centroid(x, y, w, h)

            matches.append(((x, y, w, h), centroid))

        return matches

    def \_\_call\_\_(self, frame):

        frame = frame.copy()

        fg\_mask = self.bg\_subtractor.apply(frame, None, 0.001)

        # just thresholding values

        fg\_mask[fg\_mask < 240] = 0

        fg\_mask = self.filter\_mask(fg\_mask, 0)

        return self.detect\_vehicles(fg\_mask)

cd = ContourDetection(bg\_subtractor)

bg\_subtractor = cv2.createBackgroundSubtractorMOG2(

        history=500, detectShadows=True)

# Set up image source

cap = skvideo.io.vreader(VIDEO\_SOURCE)

# skipping 500 frames to train bg subtractor

train\_bg\_subtractor(bg\_subtractor, cap, num=500)

frame = next(cap)

objects = cd(frame)

print('Getting list of [((x,y,w,h), (xc,yc)), ...]')

print(objects)

CELL6:

class PipelineRunner(object):

    def \_\_init\_\_(self, pipeline=None, log\_level=logging.INFO):

        self.pipeline = pipeline or []

        self.context = {}

        self.log = logging.getLogger(self.\_\_class\_\_.\_\_name\_\_)

        self.log.setLevel(log\_level)

        self.log\_level = log\_level

        self.set\_log\_level()

    def set\_context(self, data):

        self.context = data

    def add(self, processor):

        if not isinstance(processor, PipelineProcessor):

            raise Exception(

                'Processor should be an isinstance of PipelineProcessor.')

        processor.log.setLevel(self.log\_level)

        self.pipeline.append(processor)

    def remove(self, name):

        for i, p in enumerate(self.pipeline):

            if p.\_\_class\_\_.\_\_name\_\_ == name:

                del self.pipeline[i]

                return True

        return False

    def set\_log\_level(self):

        for p in self.pipeline:

            p.log.setLevel(self.log\_level)

    def run(self):

        for p in self.pipeline:

            self.context = p(self.context)

        self.log.debug("Frame #%d processed.", self.context['frame\_number'])

        return self.context

class PipelineProcessor(object):

    '''

        Base class for processors.

    '''

    def \_\_init\_\_(self):

        self.log = logging.getLogger(self.\_\_class\_\_.\_\_name\_\_)

CELL7:

def save\_frame(frame, file\_name, flip=True):

    # flip BGR to RGB

    if flip:

        cv2.imwrite(file\_name, np.flip(frame, 2))

    else:

        cv2.imwrite(file\_name, frame)

class ContourDetection(PipelineProcessor):

    '''

        Detecting moving objects.

    '''

    def \_\_init\_\_(self, bg\_subtractor, min\_contour\_width=35, min\_contour\_height=35, save\_image=False, image\_dir='images'):

        super(ContourDetection, self).\_\_init\_\_()

        self.bg\_subtractor = bg\_subtractor

        self.min\_contour\_width = min\_contour\_width

        self.min\_contour\_height = min\_contour\_height

        self.save\_image = save\_image

        self.image\_dir = image\_dir

    def filter\_mask(self, img, a=None):

        '''

            This filters are hand-picked just based on visual tests

        '''

        kernel = cv2.getStructuringElement(cv2.MORPH\_ELLIPSE, (2, 2))

        # Fill any small holes

        closing = cv2.morphologyEx(img, cv2.MORPH\_CLOSE, kernel)

        # Remove noise

        opening = cv2.morphologyEx(closing, cv2.MORPH\_OPEN, kernel)

        # Dilate to merge adjacent blobs

        dilation = cv2.dilate(opening, kernel, iterations=2)

        return dilation

    def detect\_vehicles(self, fg\_mask, context):

        matches = []

        # finding external contours

        contours, hierarchy = cv2.findContours(

            fg\_mask, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_TC89\_L1)

        for (i, contour) in enumerate(contours):

            (x, y, w, h) = cv2.boundingRect(contour)

            contour\_valid = (w >= self.min\_contour\_width) and (

                h >= self.min\_contour\_height)

            if not contour\_valid:

                continue

            centroid = get\_centroid(x, y, w, h)

            matches.append(((x, y, w, h), centroid))

        return matches

    def \_\_call\_\_(self, context):

        frame = context['frame'].copy()

        frame\_number = context['frame\_number']

        fg\_mask = self.bg\_subtractor.apply(frame, None, 0.001)

        # just thresholding values

        fg\_mask[fg\_mask < 240] = 0

        fg\_mask = self.filter\_mask(fg\_mask, frame\_number)

        if self.save\_image:

            save\_frame(fg\_mask, self.image\_dir +

                             "/mask\_%04d.png" % frame\_number, flip=False)

        context['objects'] = self.detect\_vehicles(fg\_mask, context)

        context['fg\_mask'] = fg\_mask

        return context

CELL8:

def distance(x, y, type='euclidian', x\_weight=1.0, y\_weight=1.0):

    if type == 'euclidian':

        return math.sqrt(float((x[0] - y[0])\*\*2) / x\_weight + float((x[1] - y[1])\*\*2) / y\_weight)

class VehicleCounter(PipelineProcessor):

    '''

        Counting vehicles that entered in exit zone.

    '''

    def \_\_init\_\_(self, exit\_masks=[], path\_size=10, max\_dst=30, x\_weight=1.0, y\_weight=1.0):

        super(VehicleCounter, self).\_\_init\_\_()

        self.exit\_masks = exit\_masks

        self.vehicle\_count = 0

        self.path\_size = path\_size

        self.pathes = []

        self.max\_dst = max\_dst

        self.x\_weight = x\_weight

        self.y\_weight = y\_weight

    def check\_exit(self, point):

        for exit\_mask in self.exit\_masks:

            try:

                if exit\_mask[point[1]][point[0]] == 255:

                    return True

            except:

                return True

        return False

    def \_\_call\_\_(self, context):

        objects = context['objects']

        context['exit\_masks'] = self.exit\_masks

        context['pathes'] = self.pathes

        context['vehicle\_count'] = self.vehicle\_count

        if not objects:

            return context

        points = np.array(objects)[:, 0:2]

        points = points.tolist()

        # add new points if pathes is empty

        if not self.pathes:

            for match in points:

                self.pathes.append([match])

        else:

            # link new points with old pathes based on minimum distance between

            # points

            new\_pathes = []

            for path in self.pathes:

                \_min = 999999

                \_match = None

                for p in points:

                    if len(path) == 1:

                        # distance from last point to current

                        d = distance(p[0], path[-1][0])

                    else:

                        # based on 2 prev points predict next point and calculate

                        # distance from predicted next point to current

                        xn = 2 \* path[-1][0][0] - path[-2][0][0]

                        yn = 2 \* path[-1][0][1] - path[-2][0][1]

                        d = distance(

                            p[0], (xn, yn),

                            x\_weight=self.x\_weight,

                            y\_weight=self.y\_weight

                        )

                    if d < \_min:

                        \_min = d

                        \_match = p

                if \_match and \_min <= self.max\_dst:

                    points.remove(\_match)

                    path.append(\_match)

                    new\_pathes.append(path)

                # do not drop path if current frame has no matches

                if \_match is None:

                    new\_pathes.append(path)

            self.pathes = new\_pathes

            # add new pathes

            if len(points):

                for p in points:

                    # do not add points that already should be counted

                    if self.check\_exit(p[1]):

                        continue

                    self.pathes.append([p])

        # save only last N points in path

        for i, \_ in enumerate(self.pathes):

            self.pathes[i] = self.pathes[i][self.path\_size \* -1:]

        # count vehicles and drop counted pathes:

        new\_pathes = []

        for i, path in enumerate(self.pathes):

            d = path[-2:]

            if (

                # need at list two points to count

                len(d) >= 2 and

                # prev point not in exit zone

                not self.check\_exit(d[0][1]) and

                # current point in exit zone

                self.check\_exit(d[1][1]) and

                # path len is bigger then min

                self.path\_size <= len(path)

            ):

                self.vehicle\_count += 1

            else:

                # prevent linking with path that already in exit zone

                add = True

                for p in path:

                    if self.check\_exit(p[1]):

                        add = False

                        break

                if add:

                    new\_pathes.append(path)

        self.pathes = new\_pathes

        context['pathes'] = self.pathes

        context['objects'] = objects

        context['vehicle\_count'] = self.vehicle\_count

        self.log.debug('#VEHICLES FOUND: %s' % self.vehicle\_count)

        return context

CELL9:

EXIT\_PTS = np.array([

    [[732, 720], [732, 590], [1280, 500], [1280, 720]],

    [[0, 400], [645, 400], [645, 0], [0, 0]]

])

SHAPE = (720,1280)

base = np.zeros(SHAPE + (3,), dtype='uint8')

exit\_mask = cv2.fillPoly(base, EXIT\_PTS, (255, 255, 255))[:, :, 0]

plt.imshow(base)

plt.show()

CELL10:

class CsvWriter(PipelineProcessor):

    def \_\_init\_\_(self, path, name, start\_time=0, fps=15):

        super(CsvWriter, self).\_\_init\_\_()

        self.fp = open(os.path.join(path, name), 'w')

        self.writer = csv.DictWriter(self.fp, fieldnames=['time', 'vehicles'])

        self.writer.writeheader()

        self.start\_time = start\_time

        self.fps = fps

        self.path = path

        self.name = name

        self.prev = None

    def \_\_call\_\_(self, context):

        frame\_number = context['frame\_number']

        count = \_count = context['vehicle\_count']

        if self.prev:

            \_count = count - self.prev

        time = ((self.start\_time + int(frame\_number / self.fps)) \* 100

                + int(100.0 / self.fps) \* (frame\_number % self.fps))

        self.writer.writerow({'time': time, 'vehicles': \_count})

        self.prev = count

        return context

BOUNDING\_BOX\_COLOUR = (255, 192, 0)

CENTROID\_COLOUR = (255, 192, 0)

CAR\_COLOURS = [(255, 192, 0)]

EXIT\_COLOR = (66, 183, 42)

class Visualizer(PipelineProcessor):

    def \_\_init\_\_(self, save\_image=True, image\_dir='images'):

        super(Visualizer, self).\_\_init\_\_()

        self.save\_image = save\_image

        self.image\_dir = image\_dir

    def check\_exit(self, point, exit\_masks=[]):

        for exit\_mask in exit\_masks:

            if exit\_mask[point[1]][point[0]] == 255:

                return True

        return False

    def draw\_pathes(self, img, pathes):

        if not img.any():

            return

        for i, path in enumerate(pathes):

            path = np.array(path)[:, 1].tolist()

            for point in path:

                cv2.circle(img, point, 2, CAR\_COLOURS[0], -1)

                cv2.polylines(img, [np.int32(path)], False, CAR\_COLOURS[0], 1)

        return img

    def draw\_boxes(self, img, pathes, exit\_masks=[]):

        for (i, match) in enumerate(pathes):

            contour, centroid = match[-1][:2]

            if self.check\_exit(centroid, exit\_masks):

                continue

            x, y, w, h = contour

            cv2.rectangle(img, (x, y), (x + w - 1, y + h - 1),

                          BOUNDING\_BOX\_COLOUR, 1)

            cv2.circle(img, centroid, 2, CENTROID\_COLOUR, -1)

        return img

    def draw\_ui(self, img, vehicle\_count, exit\_masks=[]):

        # this just add green mask with opacity to the image

        for exit\_mask in exit\_masks:

            \_img = np.zeros(img.shape, img.dtype)

            \_img[:, :] = EXIT\_COLOR

            mask = cv2.bitwise\_and(\_img, \_img, mask=exit\_mask)

            cv2.addWeighted(mask, 1, img, 1, 0, img)

        # drawing top block with counts

        cv2.rectangle(img, (0, 0), (img.shape[1], 50), (0, 0, 0), cv2.FILLED)

        cv2.putText(img, ("Vehicles passed: {total} ".format(total=vehicle\_count)), (30, 30),

                    cv2.FONT\_HERSHEY\_SIMPLEX, 0.7, (255, 255, 255), 1)

        return img

    def \_\_call\_\_(self, context):

        frame = context['frame'].copy()

        frame = np.ascontiguousarray(np.flip(frame, 2))

        frame\_number = context['frame\_number']

        pathes = context['pathes']

        exit\_masks = context['exit\_masks']

        vehicle\_count = context['vehicle\_count']

        frame = self.draw\_ui(frame, vehicle\_count, exit\_masks)

        frame = self.draw\_pathes(frame, pathes)

        frame = self.draw\_boxes(frame, pathes, exit\_masks)

        if self.save\_image:

            save\_frame(frame, self.image\_dir +

                            "/processed\_%04d.png" % frame\_number)

        context['frame'] = frame

        return context

CELL11:

# build runner

def main():

    log = logging.getLogger("main")

    # creating exit mask from points, where we will be counting our vehicles

    base = np.zeros(SHAPE + (3,), dtype='uint8')

    exit\_mask = cv2.fillPoly(base, EXIT\_PTS, (255, 255, 255))[:, :, 0]

    # there is also bgslibrary, that seems to give better BG substruction, but

    # not tested it yet

    bg\_subtractor = cv2.createBackgroundSubtractorMOG2(

        history=500, detectShadows=True)

    # processing pipline for programming conviniance

    pipeline = PipelineRunner(pipeline=[

        ContourDetection(bg\_subtractor=bg\_subtractor,

                         save\_image=True, image\_dir=IMAGE\_DIR),

        # we use y\_weight == 2.0 because traffic are moving vertically on video

        # use x\_weight == 2.0 for horizontal.

        VehicleCounter(exit\_masks=[exit\_mask], y\_weight=2.0),

        Visualizer(image\_dir=IMAGE\_DIR,save\_image=False),

        CsvWriter(path='./', name='report.csv')

    ], log\_level=logging.INFO)

    # Set up image source

    cap = skvideo.io.vreader(VIDEO\_SOURCE)

    # skipping 500 frames to train bg subtractor

    train\_bg\_subtractor(bg\_subtractor, cap, num=500)

    fourcc = cv2.VideoWriter\_fourcc(\*"MP4V")

    writer = cv2.VideoWriter(VIDEO\_OUT, fourcc, 25, (SHAPE[1], SHAPE[0]), True)

    frame\_number = -1

    for frame in cap:

        if not frame.any():

            log.error("Frame capture failed, stopping...")

            break

        frame\_number += 1

        log.info("Frame #%s" % frame\_number)

        pipeline.set\_context({

            'frame': frame,

            'frame\_number': frame\_number,

        })

        ctx = pipeline.run()

        writer.write(ctx['frame'])

        if frame\_number > PARSE\_FRAMES:

            break

    writer.release()

CELL12:  
# Parameters

# ============================================================================

IMAGE\_DIR = "./out"

VIDEO\_SOURCE = "road.mp4"

VIDEO\_OUT = "road\_parsed.mp4"

PARSE\_FRAMES = 15\*25

SHAPE = (720, 1280)  # HxW

EXIT\_PTS = np.array([

    [[732, 720], [732, 590], [1280, 500], [1280, 720]],

    [[0, 400], [645, 400], [645, 0], [0, 0]]

])

# ============================================================================

log = init\_logging()

main()

from google.colab import files

files.download('road\_parsed.mp4')