Smartathon

Complete Pothole Analysis from Visual Input

Team Name: akatsuki

Team Member: Nippun Sharma

Email: inbox.nippun@gmail.com (inbox.nippun@gmail.com)

Introduction

The problem provided is to accurately detect and localize potholes in a given area, just by using a video input stream from a moving car. The challenge itself becomes complicated as we are only allowed to use the video frames to perform any sort of predictions. In this document, I will propose an automated way for detecting, locating, and possibly reconstructing potholes just using the video input. As a proof of concept (PoC), I will also apply most of the discussed techniques on the example video file provided by the organizers.

Detection

The most straight-forward part of this challenge was to detect potholes i.e. create bounding-boxes around potholes in the video. I used a Yolov5m (medium) model to perform this task. The model was trained on the 2022 version of the Road Damage Detection Dataset. This dataset consists Potholes, Longitudinal Cracks, Transverse Cracks and Aligator Cracks.

This (https://github.com/sekilab/RoadDamageDetector) is the link to github repository of the RDD Dataset.

Tracking

In our case we also want to count the total number of unique potholes that were visible in the entire journey.
A normal detector will not provide us with unique boxes, as a pothole detected in one frame will also get detected in the next one. Thus counting the total number of detections will lead to a double-counting problem. To tackle this problem, I used detection with tracking. Tracking is the process of assigning a unique identifier to a bounding-box and keeping sure that the box has the same identifier in all subsequent frames. Tracking will prevent the double-counting and we will be able to count the actual number of potholes. I used SORT (Simple Online and Realtime Tracking), which is a computer-vision based algorithm to track bounding boxes.

In [1]:
```
from modules.config.config import config
from modules.detection.detection import apply_detection
import matplotlib.pyplot as plt

from pathlib import Path
import cv2
import pandas as pd
from tqdm import tqdm
import logging
from sort.sort import *
import torch
```

In [14]:
```
DEMO_VIDEO = "demo/sections.mov"
RESULTS_DIR = "./results"
YOLO_DIR = "./yolov5"
DETECTION_MODEL = "./pretrained_model/yolov5.onnx"

if not os.path.exists(RESULTS_DIR):
    os.mkdir(RESULTS_DIR)

if not os.path.exists(Path(RESULTS_DIR) / "detection"):
    os.mkdir(Path(RESULTS_DIR) / "detection")
```

In [5]:
```
# apply detection and tracking on the dummy video here.
# output saved in results/detection/section.mp4

cap = cv2.VideoCapture(DEMO_VIDEO)

if not cap.isOpened():
    logging.error(f"Could not open video file {DEMO_VIDEO}")
    raise

width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
fps = int(cap.get(cv2.CAP_PROP_FPS))
length = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))

out = cv2.VideoWriter((Path(RESULTS_DIR) / "detection" / f"{Path(DEMO_VIDEO).stem}.mp4").as_posix(),
    cv2.VideoWriter_fourcc(*'MP4V'), fps, (width, height))

detections_df = pd.DataFrame(columns=["id", "frame", "xmin", "ymin", "xmax", "ymax", "class"])

num_frames = 0

# object trackers.
pothole_tracker = Sort(max_age=10, min_hits=3)
pothole_ids = []

long_crack_tracker = Sort(max_age=10, min_hits=3)
long_crack_ids = []

trans_crack_tracker = Sort(max_age=10, min_hits=3)
trans_crack_ids = []

alig_crack_tracker = Sort(max_age=10, min_hits=3)
alig_crack_ids = []

model = torch.hub.load(YOLO_DIR, 'custom', DETECTION_MODEL, source='local')
pbar = tqdm(total=length)

# iterate over all video frames.
while(cap.isOpened()):
    ret, frame = cap.read()

    if ret:
        # perform detection.
        result = apply_detection(frame, model)

        if result.shape[0] == 0:
            detections = np.empty((0,5))
            pothole_tracks = pothole_tracker.update(detections)
            long_crack_tracks = long_crack_tracker.update(detections)
            trans_crack_tracks = trans_crack_tracker.update(detections)
            alig_crack_tracks = alig_crack_tracker.update(detections)
        else:
            detections = result.loc[:, ["xmin","ymin","xmax","ymax","confidence"]]
            detections[["xmin", "xmax"]] = detections[["xmin", "xmax"]] * 1920 / 640
            detections[["ymin", "ymax"]] = detections[["ymin", "ymax"]] * 1080 / 640

            potholes = detections.loc[result["name"] == "D40", :].values
            pothole_tracks = pothole_tracker.update(potholes)

            long_cracks = detections.loc[result["name"] == "D00", :].values
            long_crack_tracks = long_crack_tracker.update(long_cracks)

            trans_cracks = detections.loc[result["name"] == "D10", :].values
            trans_crack_tracks = trans_crack_tracker.update(trans_cracks)

            alig_cracks = detections.loc[result["name"] == "D20", :].values
            alig_crack_tracks = alig_crack_tracker.update(alig_cracks)

        for track in pothole_tracks:
            bbox = track[:4].astype(int)
            track_id = track[-1]

            if track_id not in pothole_ids:
                pothole_ids.append(track_id)

            detections_df.loc[len(detections_df)] = [pothole_ids.index(track_id), num_frames, track[0], track
[1], track[2], track[3], "D40"]

            cv2.rectangle(frame, bbox[:2], bbox[2:], (0,0,255), 2)
            cv2.putText(frame, f"ID: {track_id} - pothole", (bbox[0], bbox[1]-10), cv2.FONT_HERSHEY_SIMPLEX, 1.
5, (0,255,0), 2)
```

```
        for track in long_crack_tracks:
            bbox = track[:4].astype(int)
            track_id = track[-1]

            if track_id not in long_crack_ids:
                long_crack_ids.append(track_id)

            detections_df.loc[len(detections_df)] = [long_crack_ids.index(track_id), num_frames, track[0], trac
k[1], track[2], track[3], "D00"]

            cv2.rectangle(frame, bbox[:2], bbox[2:], (0,255,0), 2)
            cv2.putText(frame, f"ID: {track_id} - long. crack", (bbox[0], bbox[1]-10), cv2.FONT_HERSHEY_SIMPLE
X, 1.5, (0,255,0), 2)

        for track in trans_crack_tracks:
            bbox = track[:4].astype(int)
            track_id = track[-1]

            if track_id not in trans_crack_ids:
                trans_crack_ids.append(track_id)

            detections_df.loc[len(detections_df)] = [trans_crack_ids.index(track_id), num_frames, track[0], tra
ck[1], track[2], track[3], "D10"]

            cv2.rectangle(frame, bbox[:2], bbox[2:], (255,0,0), 2)
            cv2.putText(frame, f"ID: {track_id} - trans. crack", (bbox[0], bbox[1]-10), cv2.FONT_HERSHEY_SIMPLE
X, 1.5, (0,255,0), 2)

        for track in alig_crack_tracks:
            bbox = track[:4].astype(int)
            track_id = track[-1]

            if track_id not in alig_crack_ids:
                alig_crack_ids.append(track_id)

            detections_df.loc[len(detections_df)] = [alig_crack_ids.index(track_id), num_frames, track[0], trac
k[1], track[2], track[3], "D20"]

            cv2.rectangle(frame, bbox[:2], bbox[2:], (0,0,0), 2)
            cv2.putText(frame, f"ID: {track_id} - alig. crack", (bbox[0], bbox[1]-10), cv2.FONT_HERSHEY_SIMPLE
X, 1.5, (0,255,0), 2)

        cv2.putText(frame, f"POTHOLE COUNT: {len(pothole_ids)}", (40, 40), cv2.FONT_HERSHEY_SIMPLEX, 1.5, (0,0,
0), 2)
        cv2.putText(frame, f"LONG. CRACK COUNT: {len(long_crack_ids)}", (40, 80), cv2.FONT_HERSHEY_SIMPLEX, 1.
5, (0,0,0), 2)
        cv2.putText(frame, f"TRANS. CRACK COUNT: {len(trans_crack_ids)}", (40, 120), cv2.FONT_HERSHEY_SIMPLEX,
1.5, (0,0,0), 2)
        cv2.putText(frame, f"ALIG. CRACK COUNT: {len(alig_crack_ids)}", (40, 160), cv2.FONT_HERSHEY_SIMPLEX, 1.
5, (0,0,0), 2)

        pbar.update(1)
        num_frames += 1
        out.write(frame)
    else:
        break

pbar.close()
cap.release()
out.release()

detections_df.to_csv(Path(RESULTS_DIR) / "detection" / "detections.csv", index=False)
```

```
YOLOv5  v7.0-70-g589edc7 Python-3.8.8 torch-1.13.1 CUDA:0 (NVIDIA GeForce RTX 3060 Laptop GPU, 6144MiB)

Loading pretrained_model\yolov5.onnx for ONNX Runtime inference...
Adding AutoShape...
100%|██████████| 3844/3844 [03:21<00:00, 19.12it/s]
```

Check `results/detection` for the generated video file (`sections.mp4`) with bounding boxes and counts.

Absolute Scale

It is well known that the structure-of-motion from a single camera only results in a reconstruction up to a scale. Meaning that, there is no sense of absolute distances (such as metres or centi-meters), all points are relatively placed up to a scale. However, in our case, it is required to measure the actual dimensions of the pothole and find its absolute location within the complete journey travelled by the vehicle. Our problem falls under a special category of SFM where our vehicle puts certain constraints on the camera motion, that are known as non-holonomic constraints. Especially, when the camera is at an offset w.r.t. the car's center of gravity. The solution is based on the physics behind the instantaneous center of rotation (ICR). Basically, any moving object can be considere as rotating about its ICR.

For an in-depth analysis into the setup and solution, you can read this very interesting paper (https://rpg.ifi.uzh.ch/docs/ICCV09_scaramuzza.pdf) by Davide Scaramuzza. Below is the code that I have written after reading the paper and I use it for generating an approximate absolute scale value. I have used the least-squares version, which is a 3-point algorithm. Also, as a quick demonstration (when we plot using visual odometry) I have extracted a subset of frames from the demo video where the car is turning around the corner.

In [3]:

```
def find_theta_phi(image_1_pts, image_2_pts):
    # find theta and phi angles from image correspondences.
    # make sure that there are at-least 3 corresponding pairs.

    N = image_1_pts.shape[0]
    A = np.zeros((N,4), dtype=float)

    A[:,0] = image_1_pts[:,0] * image_2_pts[:,1]
    A[:,1] = image_1_pts[:,1] * image_2_pts[:,0]
    A[:,2] = image_1_pts[:,2] * image_2_pts[:,1]
    A[:,3] = image_1_pts[:,1] * image_2_pts[:,2]

    U, S, V = np.linalg.svd(A)
    result = V[:,-1]

    phi = np.arctan2(result[2], -result[0])
    theta = phi + np.arctan2(result[3], result[1])
    return theta, phi

def normalize_point(point):
    # normalize a point about a sphere of radius 1.
    R = 1

    xAvg = point[:,0].mean()
    yAvg = point[:,1].mean()
    xy_norm = (((point - np.array([[xAvg, yAvg]])) ** 2).sum(axis=1) ** 0.5).mean()
    diagonal_element = (R ** 0.5) / xy_norm
    element_13 = -(R ** 0.5) * xAvg / xy_norm
    element_23 = -(R ** 0.5) * yAvg / xy_norm
    norm_mat = np.array([[diagonal_element, 0, element_13], [0, diagonal_element, element_23], [0, 0, 1]])
    point = np.concatenate([point, np.ones((point.shape[0],1))], axis=1)
    return norm_mat.dot(point.T).T

def absolute_scale(images):
    # maximum frames to lookahead to find a valid match.
    MAX_LOOKAHEAD = 15

    # threshold for theta.
    THETA_THRES = 30

    # this is the offset value of the camera from the Center of Gravity of the car.
    # I have taken the same value as is taken in paper.
    # The value is in meters.
    D_COM = 0.9

    left = 0
    right = 1
    N = len(images)
    curvatures = []

    pbar = tqdm(total=N)

    # generate a list of valid curvatures.
    while(right < N):
        img1 = cv2.cvtColor(images[left], cv2.COLOR_RGB2GRAY)
        img2 = cv2.cvtColor(images[right], cv2.COLOR_RGB2GRAY)

        orb = cv2.ORB_create(3000)
        FLANN_INDEX_LSH = 6
        index_params = dict(algorithm=FLANN_INDEX_LSH, table_number=6, key_size=12, multi_probe_level=1)
        search_params = dict(checks=50)
        flann = cv2.FlannBasedMatcher(indexParams=index_params, searchParams=search_params)

        kp1, des1 = orb.detectAndCompute(img1, None)
        kp2, des2 = orb.detectAndCompute(img2, None)

        matches = flann.knnMatch(des1, des2, k=2)

        # Find the matches there do not have a to high distance
        good = []
        try:
            for m, n in matches:
                if m.distance < 0.8 * n.distance:
                    good.append(m)
        except ValueError:
            pass

        # Get the image points form the good matches
        q1 = np.float32([kp1[m.queryIdx].pt for m in good])
        q2 = np.float32([kp2[m.trainIdx].pt for m in good])
```

```
        # normalizing points about the sphere of radius 1.
        q1 = normalize_point(q1)
        q2 = normalize_point(q2)

        theta, phi = find_theta_phi(q1, q2)
        theta_deg = theta * 180 / np.pi

        if abs(theta_deg) < THETA_THRES:
            # no motion detected.
            right += 1
            pbar.update(1)
            continue

        rho = (D_COM * np.sin(phi) - D_COM * np.sin(phi - theta)) / np.sin(phi - theta/2.)

        if rho < 0:
            left = right
            right += 1
            pbar.update(1)
            continue

        k = 2 * np.sin(theta / 2.) / rho
        curvatures.append({
            "value": k,
            "left": left,
            "right": right,
            "rho": rho
        })

        left = right
        right += 1
        pbar.update(1)

    print(f"A total of {len(curvatures)} curvatures were extracted.")

    # now we select which curvatures depict a circular motion.
    mask = []
    for i in range(len(curvatures)-1):
        val = abs(curvatures[i]["value"] - curvatures[i+1]["value"]) / abs(curvatures[i]["value"])
        if val < 0.1:
            mask.append(1)
        else:
            mask.append(0)

    circulars = []
    for i in range(len(mask)):
        if mask[i] == 1:
            circulars.append(curvatures[i])

    print(f"A total of {len(circulars)} circular motions were extracted.")

    # create a list of possible absolute scales and plot a histogram.
    rhos = []
    for circ in circulars:
        rho = circ["rho"]
        rhos.append(rho)

    return rhos, circulars
```

In [4]:
```
# reading the subset frames present at 'demo/turn/'
img_path = "demo/turn/images/"
image_paths = sorted(os.listdir(img_path))
images = []
for path in image_paths:
    img = cv2.imread(img_path + "/" + path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    images.append(img)
```

In [5]:
```
rhos, circulars = absolute_scale(images)
```

```
100%|██████████| 690/691 [01:40<00:00,  6.86it/s]

A total of 354 curvatures were extracted.
A total of 15 circular motions were extracted.
```

In [6]:
```
# creating a histogram of rhos.
counts, bins = np.histogram(rhos, bins=5)

# now we extract the bin value with highest frequency.
# note that this just a heuristic to remove the affect
# of possible outliers / wrong predictions. this method
# can definitely be improved by using some more robust
# techniques.
idxs = np.argsort(counts)
rho_prediction = bins[idxs[-1]]
print("Predicted absolute scale:", rho_prediction)
```

```
Predicted absolute scale: 0.27965190323954897
```

Visual Odometry

Visual Odometry is the process of reconstructing the camera path through visual input / video. The VO is calculated in a relative scal
However, since we have estimated an absolute scale we have the information to construct an odometry in the absolute scale.

In [31]:
```
from modules.odometry.visual_odometry import VisualOdometry
import io
```

In [9]:
```
odometry = VisualOdometry("demo/turn", rho_prediction)
estimated_path = []

if not os.path.exists(Path(RESULTS_DIR) / "odometry"):
    os.mkdir(Path(RESULTS_DIR) / "odometry")

pbar = tqdm(total=len(images))

for i in range(len(images)):
    if i == 0:
        curr_pose = np.eye(4)
    else:
        q1, q2 = odometry.get_matches(i)
        transf, _, _, _ = odometry.get_pose(q1, q2)
        curr_pose = np.matmul(curr_pose, np.linalg.inv(transf))
        estimated_path.append((curr_pose[0, 3], curr_pose[2, 3]))
    pbar.update(1)
```

```
100%|██████████| 691/691 [01:12<00:00, 10.09it/s]
```

In [14]:
```
# plot the absolute scaled trajectory.
estimated_path = np.array(estimated_path)
fig = plt.figure()
plt.title("Estimated Trajectory around corner.")
plt.scatter(estimated_path[:,0], estimated_path[:,1])
# TODO: plot circulars.
plt.gca().set_xlim(-100, 100)
plt.gca().set_ylim(-100, 100)
plt.grid("minor")
plt.xlabel("x (in meters)")
plt.ylabel("y (in meters)")
plt.savefig("results/odometry/vo.png")
```

The generated trajectory is saved in `results/odometry/vo.png` . Also, take a look at the images present in `demo/turn/images` for getting an idea about the actual trajectory. As we can observe, the actual trajectory (as observed visually) is very similar to the estimated one. In fact, the scale of the estimated trajectory is absolute. This means that we can exactly measure the location (in meters) of the car at any point in its journey. Also, we can correlate this frame-by-frame data along with the detection and tracking data obtained earlier and predict the exact coordinates of a particular pothole, thus flagging it efficiently.

We can also provide this functionality in real-time by using a faster algorithm. Again, this paper (https://drops.dagstuhl.de/opus/volltexte/2011/2950/pdf/10371.ScaramuzzaDavide.Paper.2950.pdf) by David Scaramuzza can prove to be useful in this case. He solves the problem of visual odometry in real-time, using a 1-point RANSAC algorithm that exploits the non-holonomic constraints of the car.

3D Reconstruction

The final part of the pipeline can be to create a 3 dimensional reconstruction of a pothole. Structure From Motion is the most popular technique that is used to create 3D point clouds using multiple photos. However, our problem is of a special kind. We have to model the road surface which is a near-planar surface. Due to this planarity, an ambiguity arises in the calculation of the Fundamental Matrix. Therefore, the normal equations of SFM might not produce great results in our case.

To know more about how near-planar surfaces cause this ambiguity, this paper (https://www.mdpi.com/1424-8220/20/6/1640) proves the same. The paper also derives a new unambiguous equation for calculating the fundamental matrix using the homography matrix. Further, this paper also provides a great recursive post-processing technique on pothole point clouds that can improve the structure of the reconstruction even more. Unfortunately due to time-constraint, I was unable to implement this paper on the given demo video.

So, I used a software known as VisualSFM for dense point-cloud reconstruction for different images of the same pothole. This was comparitively easy and less complex as I was already detecting and tracking the potholes. So, using the bounding boxes I extracted the pothole (whose 3d reconstruction is to be done) from all the frames in which it was tracked. This lead to a collection of different images for every pothole, from different viewing angles.

In [50]:
```
detections_df = pd.read_csv("results/detection/detections.csv")
detections_df.head()
```

Out[50]:

	id	frame	xmin	ymin	xmax	ymax	class
0	5043.0	89	285.194881	640.341441	508.731961	779.049773	D20
1	5049.0	105	1222.791426	535.750690	1329.495472	579.264288	D40
2	5049.0	106	1233.636628	545.713590	1343.380073	590.756086	D40
3	5050.0	107	-20.779910	602.979566	489.229553	1005.932807	D40
4	5049.0	107	1247.344331	555.332061	1359.689536	601.389324	D40

In [35]:
```
POTHOLE_CLASS = "D40"
POTHOLE_ID = 7

pothole_12 = detections_df.loc[(detections_df["id"] == POTHOLE_ID) & (detections_df["class"] == POTHOLE_CLASS),
:]
pothole_12
```

Out[35]:

	id	frame	xmin	ymin	xmax	ymax	class
43	7	125	639.380196	539.716118	833.364392	582.382895	D40
50	7	126	611.615343	547.548653	824.376084	595.556804	D40
58	7	127	586.161660	552.729840	822.513836	608.677686	D40
66	7	128	572.135991	563.758007	813.180840	623.176119	D40
74	7	129	555.808030	578.041087	812.014631	643.019943	D40
82	7	130	538.208653	593.624555	807.808649	663.475136	D40
90	7	131	547.344992	620.533102	802.444096	687.079139	D40
97	7	132	529.844143	642.336836	785.561134	710.263165	D40
104	7	133	514.626662	666.427805	773.031477	736.881315	D40
111	7	134	487.764412	693.102580	755.764575	767.769809	D40
117	7	135	446.024936	720.749031	732.852272	803.554656	D40
123	7	136	399.812903	756.784818	706.883955	847.776911	D40
129	7	137	343.306560	798.885971	676.702865	901.491118	D40

In [38]:
```
pothole_crops = []

# relax bbox to also get some surroundings.
relax = 40

cap = cv2.VideoCapture(DEMO_VIDEO)

if not cap.isOpened():
    raise RuntimeError("Could not read the video!!!")

num_frames = 0
length = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
pbar = tqdm(total=length)

while cap.isOpened():
    ret, frame = cap.read()

    if ret:
        if num_frames in pothole_12.loc[:, "frame"].to_list():
            bbox = pothole_12.loc[pothole_12["frame"] == num_frames, ["xmin", "ymin", "xmax", "ymax"]]
            xmin, xmax, ymin, ymax = int(bbox["xmin"]), int(bbox["xmax"]), int(bbox["ymin"]), int(bbox["ymax"])
            pothole_cropped = img[xmin-relax:xmax+relax, ymin-relax:ymax+relax].copy()
            pothole_crops.append(pothole_cropped)

        num_frames += 1
    else:
        break

    pbar.update(1)

cap.release()
```

```
100%|██████████| 3844/3844 [03:04<00:00, 20.87it/s]
 99%|█████████▉| 3824/3844 [00:17<00:00, 222.69it/s]
```

In [40]:
```
if not os.path.exists(Path(RESULTS_DIR) / "reconstruction"):
    os.mkdir(Path(RESULTS_DIR) / "reconstruction")

if not os.path.exists(Path(RESULTS_DIR) / "reconstruction" / "images"):
    os.mkdir(Path(RESULTS_DIR) / "reconstruction" / "images")

for idx,crop in enumerate(pothole_crops):
    cv2.imwrite((Path(RESULTS_DIR) / "reconstruction" / "images" / f"{idx}.png").as_posix(), crop)
```

The resulting cropped images can then be used with VisualSFM or any other SFM tool.

Conclusion

As we observed that, even by using a single video source, we can generate a lot of analysis about the surrounding. The detection aspect of this approach is definitely more mature. We can even say that the accuracy obtained through pothole detection and tracking can be at least as good as LIDAR if not better. Moreover, the overall infrastructure required to setup this is negligible and this can even be deployed on smartphones. The solution is completely autonomous as there is no input required from a human. From object detection and tracking to 3d reconstruction, everything can be done automatically using different heuristics. A human is needed only in a supervising capacity.