computer_vision_research

Predicting trajectories of objects

Abstract

TODO: Abstract

TODO: implement SQLite DB logging

Notice: linear regression implemented, very primitive, but working
Notice: tracking could be improved: calculating the average of bounging box area, x, y, width, height or iterate trough the history for a given depth
Notice: To tell which direction is the object moving is very tricky, made a quick function to tell its in main.py

Darknet

For detection, I used darknet neural net and YOLOV4 pretrained model. [1] In order to be able to use the darknet api, build from source with the LIB flag on. Then copy libdarknet.so to root dir of the project. (My Makefile to build darknet can be found in the darknet_config_files directory)

Notice: Using the yolov4-csp-x-swish.cfg and weights with RTX 3070 TI is doing 26 FPS with 69.9% precision, this is the most stable detection so far, good base for tracking and predicting

For Darknet, I wrote an API hldnapi.py, that makes object detection more easier. cvimg2detections(img) it takes only an opencv img and returns the detections in format [label, confidence, xywh]

YOLOV7

Yolov7 is the most recent version of YOLO. Darknet is no more, the source code of the neural net is in PyTorch. Original-Repository [2]. To work with my framework, I read the whole codebase of Yolov7. I wrote yolov7api.py, function load_model(device, weights, imgsz, classify) can load the desired yolo model, if GPU is used half precision can be used (FP16 instead of FP32), detect(img) takes an opencv image as argument, it can take a lot more arguments, but those are only for parametization, there are default values set for those arguments, that are tested. The image has to be resized to the size of the NeuralNet. After the model is loaded, we can input the resized image to the neural net. The results are a matrix shaped (number of input images, number of detections, 6). A detection is a vector of [x, y, x, y, confidence, class] (first xy is top-left, second xy is bottom-right). The raw output of the neural net has to be resized to fit the original image. The output is still not good for my framework. The output have to be converted to a matrix of shape(number of detections, 3) what is looks like [label, confidence, (xywh)] xywh is center xy coordinates and width, height of bbox.

NOTICE: If pytorch throws this error: RuntimeError: CUDA out of memory. Tried to allocate X.XX GiB (GPU 0; X.XX GiB total capacity; X.XX MiB already allocated; X.XX GiB free; X.XX GiB reserved in total by PyTorch) If reserved memory is >> allocated memory try setting max_split_size_mb to avoid fragmentation. See documentation for Memory Management and PYTORCH_CUDA_ALLOC_CONF. Then set environment variable PYTORCH_CUDA_ALLOC_CONF to `PYTORCH_CUDA_ALLOC_CONF="max_split_size_mb:256"`, if this does not solve the problem, play with the `max_split_size_mb`, try to give it other sizes.

Installation

Download yolov7 weights file from yolov7.pt, then copy or move it to yolov7 directory.

Create conda environment and add yolov7 to PYTHONPATH.

```
conda create -n <insert name here> python=3.9
conda install pytorch torchvision torchaudio cudatoolkit=11.6 opencv
matplotlib pandas tqdm pyyaml seaborn -c conda-forge -c pytorch
export PYTHONPATH="${PYTHONPATH}:<PATH to YOLOV7 directory>"
```

The setup of PYTHONPATH variable is very important, becouse python will throw a module error. To not have to set this environment variable every time use `conda env config vars set PYTHONPATH=${PYTHONPATH}:<PATH to YOLOV7 directory>"` command.

In case this is not working, I implemented a gpu memory freeing function, which is called when yolov7 is imported or yolov7 model is loaded.

Tracking of detected objects

Base idea: track objects from one frame to the other, based on x and y center coordinates. This solution require very minimal resources.

Euclidean distances: This should be more precise, but require a lot more computation. Have to examine this technique further to get better results.

Deep-SORT: Simple Online and Realtime Tracking with convolutonal neural network. Pretty much based on Kalmanfilter. See the arXiv preprint for more information. [3]

Determining wheter an object moving or not

Temporary solution: Calculating the tracking history's last and the first detection's euclidean distance.

```
self.isMoving = ((self.history[0].X-self.history[-1].X)**2 +
(self.history[0].Y-self.history[-1].Y)**2)**(1/2) > 7.0
```

Throw away old detections and trackings

This can save read, write time and memory.

HistoryDepth: Implemented a historyDepth variable, that determines how long back in time should we track an objects detection data. With this, we can throw away old trackings if they are not on screen any more.

Predicting trajectories of moving objects

Linear Regression

Using **Scikit Learn Linear Models**

```
model =
linear_model.RANSACRegressor(base_estimator=linear_model.LinearRegression()
, random_state=30, min_samples=X_train.reshape(-1,1).shape[1]+1)
reg = model.fit(X_train.reshape(-1,1), y_train.reshape(-1,1))
y_pred = reg.predict(X_test.reshape(-1,1))
```

Best working linear model RANSACRegressor() with base_estimator LinearRegression().

TODO: this has to implemented, calculate weights based on detecions position.

Polynom fitting

Using Sklearn PolynomialFeatures function to generate X and Y training points for the estimator.

The PolynomialFeatures and the estimator have to be inputted to the make_pipeline function.

```
polyModel = make_pipeline(PolynomialFeatures(degree),
linear_model.RANSACRegressor(base_estimator=linear_model.Ridge(alpha=0.5),
random_state=30, min_samples=X_train.reshape(-1,1).shape[1]+1))
polyModel.fit(X_train.reshape(-1, 1), y_train.reshape(-1, 1))
y_pred = polyModel.predict(X_test.reshape(-1, 1))
```

Spline

TODO: Implement Spline, not working yet.

Regression with coordinate depending weigths

Kalman filter calculates velocities

Global heatmap of traffic

TODO: Clustering, KNN <- Scikit Learn **Clustering Algorithm**: Affinity Propagation. (**NOTICE**: This algorithm seems to give nonsense results, will have to test other ones too.)

To make the predictions smarter, a learning algorithm have to be implemented, that trains on the detection and prediction history.

NOTICE: New idea, gather detections, that velocity vector points in the same direction.

Documentation

1. Building main loop of the program to be able to input video sources, using OpenCV VideoCapture. From VideoCapture object frames can be read. `cv.imshow("FRAME", frame)` imshow function opens GUI window to show actual frame.

```
    cap = cv.VideoCapture(input)
    # exit if video cant be opened
    if not cap.isOpened():
        print("Source cannot be opened.")
        exit(0)
    .
    .
    .
    while(1):
      ret, frame = cap.read()
      if frame is None:
          break

    cv.imshow("FRAME", frame)
    if cv.waitKey(1) == ord('p'):
        if cv.waitKey(0) == ord('r'):
            continue
    if cv.waitKey(10) == ord('q'):
            break
```

2. Implement YOLO API - hldnapi.py - that works with the C-API of Darknet. In this function, the image has to be transformed to Darknet be able to run inference on it. `cv.cvtColor(image, cv.COLOR_BGR2RGB)` convert OpenCV color (Blue,Green,Red) to Darknet color (Red, Green, Blue). `cv.resize(image_rgb, (darknet_width, darknet_height), interpolation=cv.INTER_LINEAR)` resize image to Darknet's neural net image size. `darknet.detect_image(network, class_name, img_for_detect)` run detection on preprocessed image. This function returns a tuple (label, confidence, bbox[x,y,w,h]), the bounding box coordinates have to be resized to the original image.

```
    def cvimg2detections(image):
        """Fcuntion to make it easy to use darknet with opencv

        Args:
            image (Opencv image): input image to run darknet on

        Returns:
            detections(tuple): detected objects on input image (label,
confidence, bbox(x,y,w,h))
        """
        # Convert frame color from BGR to RGB
        image_rgb = cv.cvtColor(image, cv.COLOR_BGR2RGB)
        # Resize image for darknet
        image_resized = cv.resize(image_rgb, (darknet_width,
darknet_height), interpolation=cv.INTER_LINEAR)
        # Create darknet image
        img_for_detect = darknet.make_image(darknet_width, darknet_height,
3)
        # Convert cv2 image to darknet image format
        darknet.copy_image_from_bytes(img_for_detect,
image_resized.tobytes())
```

```
        # Load image into nn and get detections
        detections = darknet.detect_image(network, class_names,
img_for_detect)
        darknet.free_image(img_for_detect)
        # Resize bounding boxes for original frame
        detections_adjusted = []
        for label, confidence, bbox in detections:
            bbox_adjusted = convert2original(image, bbox)
            detections_adjusted.append((str(label), confidence,
bbox_adjusted))
        return detections_adjusted
```

3. Implement classes for storing the detections and object trackings. The classes dont have to be overly complex, they must be easy to read and understand. A `class Detection()` and a `class TrackedObject()` was created. The implementation can be found in the historyClass.py file. Detection class has 7 attributes, label, confidence, X, Y, Width, Height, frameID. TrackedObject class has 11, objID, label, futureX, futureY, history, isMoving, time_since_update, max_age, mean, X, Y, VX, VY.

4. Iplement object tracking algorithm. Base idea was to calculate x and y coordinate distances between detection objects. This is a very primitive way of tracking, for initial testing it was good, but I had to find a more accurate tracking algorithm.

5. First prediction algorithm with scikit-learn's LinearRegression function library. The predictLinear() function takes 3 arguments, a trackedObject object from historyClass.py, historyDepth to determine, how big is the learning set, and a futureDepth to know how far in the future to predict. To do the regression, at least 3 detections should occur. With the k variable we can tell the LinearRegression algorithm, on how many points from the training set to train on. Before running the regression, the movementIsRight() function determines wheter the object moving right or left, this is crucial in generation of the prediction points. After we run the regression, the futureX and futureY vector of the trackedObject object can be updated with the predicted values. For the regression I use the simple Ordinary Least Squares (OLS) method. Linear regression formula: $$\hat{y} (w, x) = w_0 + w_1 x_1 + ... + w_p x_p$$ Ordinary Least Squares formula: $$\min_{w} || X w - y||_2^2$$

```
def movementIsRight(obj: TrackedObject):
    """Returns true, if the object moving right, false otherwise.

    Args:
        obj (TrackedObject): tracking data of an object

    Return:
        bool: Tru if obj moving right.

    """
    return obj.VX > 0

def predictLinear(trackedObject: TrackedObject, k=3, historyDepth=3,
futureDepth=30):
    """Fit linear function on detection history of an object, to predict
```

```
future coordinates.

    Args:
        trackedObject (TrackedObject): The object, which's future
coordinates should be predicted.
        k (int, optional): Number of training points, ex.: if historyDepth
is 30 and k is 3, then the 1st, 15th and 30th points will be training
points. Defaults to 3.
        historyDepth (int, optional): Training history length. Defaults to
3.
        futureDepth (int, optional): Prediction vectors length. Defaults to
30.
    """
    x_history = [det.X for det in trackedObject.history]
    y_history = [det.Y for det in trackedObject.history]
    if len(x_history) >= 3 and len(y_history) >= 3:
        # k (int) : number of training points
        # k = len(trackedObject.history)
        # calculating even slices to pick k points to fit linear model on
        slice = len(trackedObject.history) // k
        X_train = np.array([x for x in x_history[-historyDepth:-1:slice]])
        y_train = np.array([y for y in y_history[-historyDepth:-1:slice]])
        # check if the movement is right or left, becouse the generated
x_test vector
        # if movement is right vector is ascending, otherwise descending
        if movementIsRight(trackedObject):
            X_test = np.linspace(X_train[-1], X_train[-1]+futureDepth)
        else:
            X_test = np.linspace(X_train[-1], X_train[-1]-futureDepth)
        # fit linear model on the x_train vectors points
        model = linear_model.LinearRegression(n_jobs=-1)
        reg = model.fit(X_train.reshape(-1,1), y_train.reshape(-1,1))
        y_pred = reg.predict(X_test.reshape(-1,1))
        trackedObject.futureX = X_test
        trackedObject.futureY = y_pred
```

6. Integrating Deep-SORT tracking into the program. Kalman filter and CNN that has been trained to discriminate pedestrians on a large-scale person re-identification dataset. [3] The Kalman filter implementation uses 8 dimensional space (x, y, a, h, vx, vy, va, vh) to track objects.

7. Prediction with Polynom fitting using Scikit-Learn's PolynomTransformer. This is similar to the Linear fitting, but this makes it possible to predict curves in an objects trajectory based on the object's position history. The only difference between the predictLinear and this algorithm, that a PolynomTransformer transforms the history data.

```
def predictPoly(trackedObject: TrackedObject, degree=3, k=3,
historyDepth=3, futureDepth=30):
    """Fit polynomial function on detection history of an object, to
predict future coordinates.

    Args:
        trackedObject (TrackedObject): The object, which's future
coordinates should be predicted.
        degree (int, optional): The polynomial functions degree. Defaults
to 3.
        k (int, optional): Number of training points, ex.: if historyDepth
is 30 and k is 3, then the 1st, 15th and 30th points will be training
points. Defaults to 3.
        historyDepth (int, optional): Training history length. Defaults to
3.
        futureDepth (int, optional): Prediction vectors length. Defaults to
30.
    """
    x_history = [det.X for det in trackedObject.history]
    y_history = [det.Y for det in trackedObject.history]
    if len(x_history) >= 3 and len(y_history) >= 3:
        # k (int) : number of training points
        # k = len(trackedObject.history)
        # calculating even slices to pick k points to fit linear model on
        slice = len(trackedObject.history) // k
        X_train = np.array([x for x in x_history[-historyDepth:-1:slice]])
        y_train = np.array([y for y in y_history[-historyDepth:-1:slice]])
        # generating future points
        if movementIsRight(trackedObject):
            X_test = np.linspace(X_train[-1], X_train[-1]+futureDepth)
        else:
            X_test = np.linspace(X_train[-1], X_train[-1]-futureDepth)
        # poly features
        polyModel = make_pipeline(PolynomialFeatures(degree),
linear_model.Ridge(alpha=1e-3))
        polyModel.fit(X_train.reshape(-1, 1), y_train.reshape(-1, 1))
        # print(X_train.shape, y_train.shape)
        y_pred = polyModel.predict(X_test.reshape(-1, 1))
        trackedObject.futureX = X_test
        trackedObject.futureY = y_pred
```

8. Prediction with splines using Scikit-Learn's SplineTransformer. Spline can only be fitted on data we have, so it cant predict on its own. Before fitting spline on any data, polynom fitting should be done first, then on the result data we can fit a spline curve.

```
# TODO
```

9. Implement database logging, to save results for later analyzing. The init_db(video_name: str) function creates the database. Name of the video, that is being played, will be the name of the database with a .db appended at the end of it. After the database file is created, schema script will be executed.
   This is the schema of the database.

```
CREATE TABLE IF NOT EXISTS objects (
    objID INTEGER PRIMARY KEY NOT NULL,
    label TEXT NOT NULL
);
CREATE TABLE IF NOT EXISTS detections (
                objID INTEGER NOT NULL,
                frameNum INTEGER NOT NULL,
                confidence REAL NOT NULL,
                x REAL NOT NULL,
                y REAL NOT NULL,
                width REAL NOT NULL,
                height REAL NOT NULL,
                vx REAL NOT NULL,
                vy REAL NOT NULL,
                ax REAL NOT NULL,
                ay REAL NOT NULL,
                FOREIGN KEY(objID) REFERENCES objects(objID)
            );
CREATE TABLE IF NOT EXISTS predictions (
                objID INTEGER NOT NULL,
                frameNum INTEGER NOT NULL,
                idx INTEGER NOT NULL,
                x REAL NOT NULL,
                y REAL NOT NULL
            );
CREATE TABLE IF NOT EXISTS metadata (
                historyDepth INTEGER NOT NULL,
                futureDepth INTEGER NOT NULL,
                yoloVersion TEXT NOT NULL,
                device TEXT NOT NULL,
                imgsize INTEGER NOT NULL,
                stride INTEGER NOT NULL,
                confidence_threshold REAL NOT NULL,
                iou_threshold REAL NOT NULL
            );
CREATE TABLE IF NOT EXISTS regression (
                linearFunction TEXT NOT NULL,
                polynomFunction TEXT NOT NULL,
                polynomDegree INTEGER NOT NULL,
                trainingPoints INTEGER NOT NULL
);
```

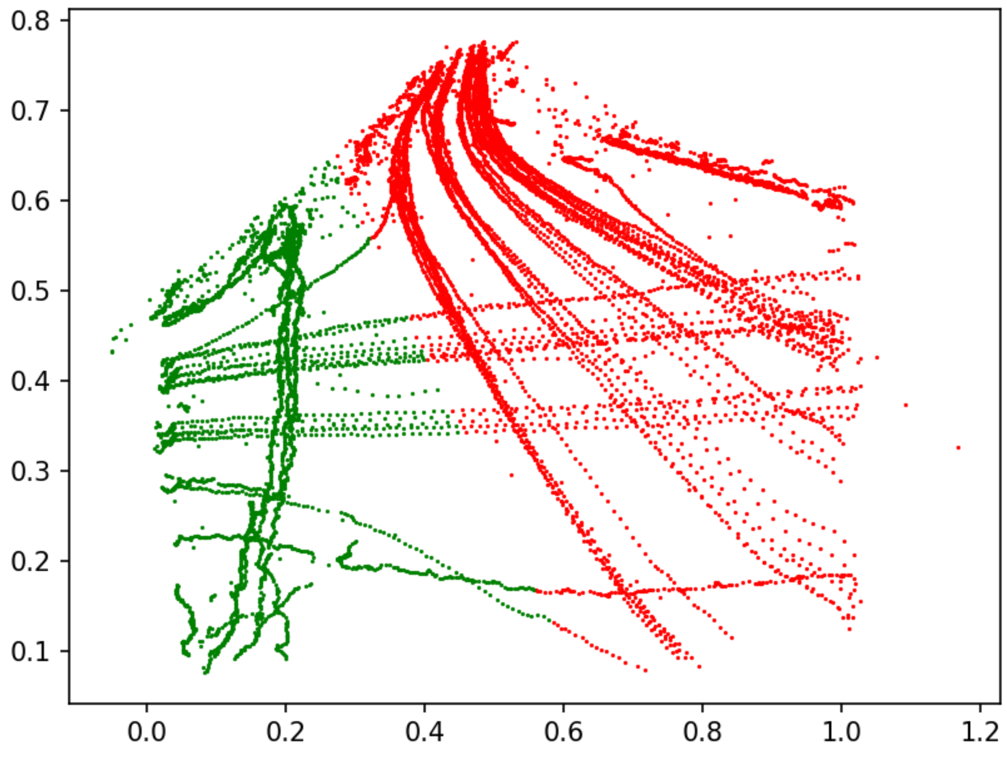
Every object is stored in the objects table, objID as primary key, will help us identify detections. Detections are stored in the detections table, here the objID is a foreign key, that tells us which detection belongs to which object. Predictions have an own table, to a single frame and a single object there can be multiple predictions. THe program's inner environment is also being logged as metadata, historyDepth is the length of the training set. FutureDepth is the length of the prediction vector. Yolo version is also being logged, becouse of the legacy version 4 (although yolov4 is not really used anymore, it is just an option, that propably will be taken out), imgsize is the input image size of the neural network, stride is how many pixels the convolutonal filter slides over the image. Confidence threshold and iou threshold will determine which

detection of yolo will we accept, if the propability of a detection being right. To the regression table, will be the regression function's configuration values stored.
TODO: detection logging is slowing down the program, find a way to make it faster

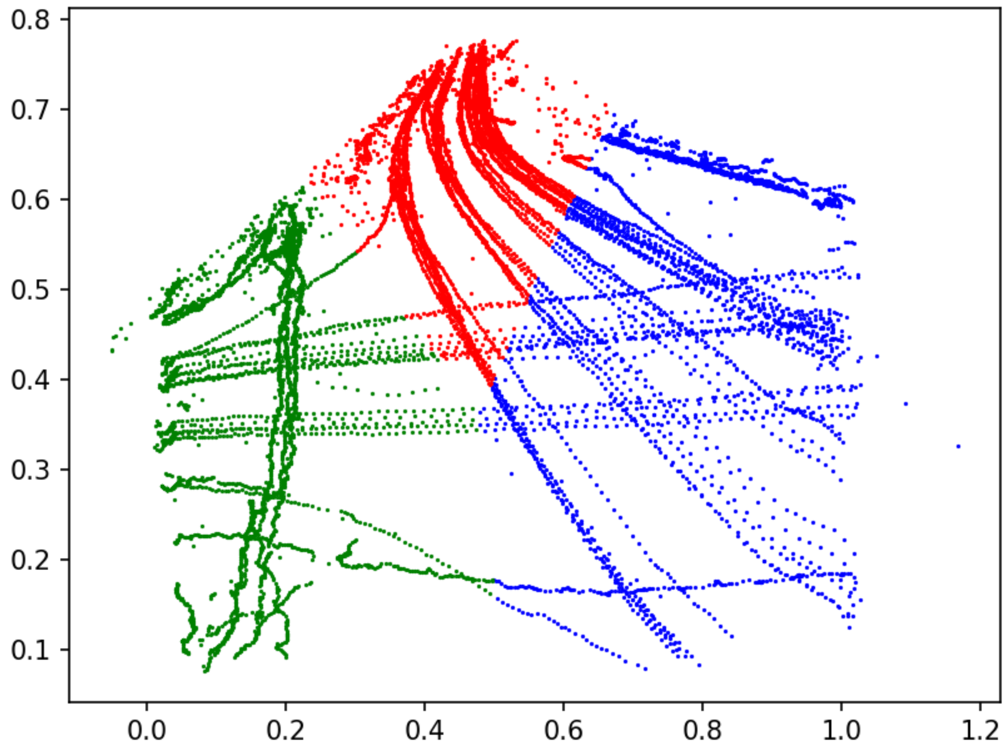
10. The logging makes it possible, to analyze the data without running the videos each time. For this, data loading functions are needed, that fetches the resutls from the database. These functions are implemented in the databaseLoader.py script. Each function returns a list of all entries logged in the database.

11. Next step after data loading module, is to create heatmap of the traffic data logged from videos. For better visuals, each object has its own coloring, so it also shows, how good DeepSort algorithm works.

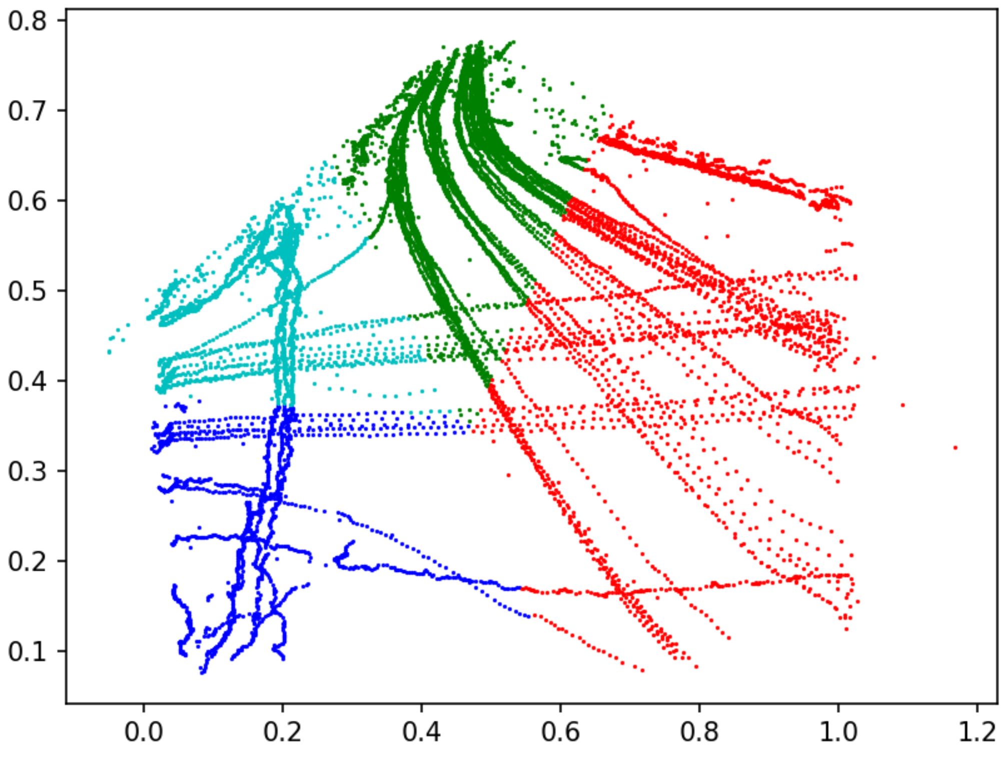
12. With scikit-learn's clustering module, clusters from the gathered data can be created. The point of this, is when a crossroad being observed, the paths can be identified, with this knowledge, personalised training can be done for each scenario. For first k_means algorithm was tested. The KMeans algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the inertia or within-cluster sum-of-squares (see below). This algorithm requires the number of clusters to be specified. It scales well to large numbers of samples and has been used across a large range of application areas in many different fields. The k-means algorithm divides a set of N samples X into K disjoint clusters C, each described by the mean of the samples in the cluster. The means are commonly called the cluster “centroids”; note that they are not, in general, points from X, although they live in the same space. The K-means algorithm aims to choose centroids that minimise the inertia, or within-cluster sum-of-squares criterion: $$\sum_{i=0}^{n}\min_{\mu_j \in C}(||x_i - \mu_j||^2)$$ Although this algorithm does not require that much computation, cant identify lanes on a crossroad. The result plots can be found in dir "research_data/sherbrooke_video/".



k_means algorithm with 2 initial cluster



k_means algorithm with 3 initial cluster



k_means algorithm with 4 initial cluster

References

Darknet-YOLO

[1]
@misc{bochkovskiy2020yolov4,
title={YOLOv4: Optimal Speed and Accuracy of Object Detection},
author={Alexey Bochkovskiy and Chien-Yao Wang and Hong-Yuan Mark Liao},
year={2020},
eprint={2004.10934},
archivePrefix={arXiv},
primaryClass={cs.CV}
}
@InProceedings{Wang_2021_CVPR,
author = {Wang, Chien-Yao and Bochkovskiy, Alexey and Liao, Hong-Yuan Mark},
title = {{Scaled-YOLOv4}: Scaling Cross Stage Partial Network},
booktitle = {Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)},
month = {June},
year = {2021},
pages = {13029-13038}
}

[2] @article{wang2022yolov7,
title={{YOLOv7}: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors},
author={Wang, Chien-Yao and Bochkovskiy, Alexey and Liao, Hong-Yuan Mark},

journal={arXiv preprint arXiv:2207.02696},
year={2022}
}

DeepSORT

[3]
@inproceedings{Wojke2017simple,
title={Simple Online and Realtime Tracking with a Deep Association Metric},
author={Wojke, Nicolai and Bewley, Alex and Paulus, Dietrich},
booktitle={2017 IEEE International Conference on Image Processing (ICIP)},
year={2017},
pages={3645--3649},
organization={IEEE},
doi={10.1109/ICIP.2017.8296962}
}
@inproceedings{Wojke2018deep,
title={Deep Cosine Metric Learning for Person Re-identification},
author={Wojke, Nicolai and Bewley, Alex},
booktitle={2018 IEEE Winter Conference on Applications of Computer Vision (WACV)},
year={2018},
pages={748--756},
organization={IEEE},
doi={10.1109/WACV.2018.00087}
}