



SDAV: Camera Perception SmartData for Lane Tracking

1. Summary

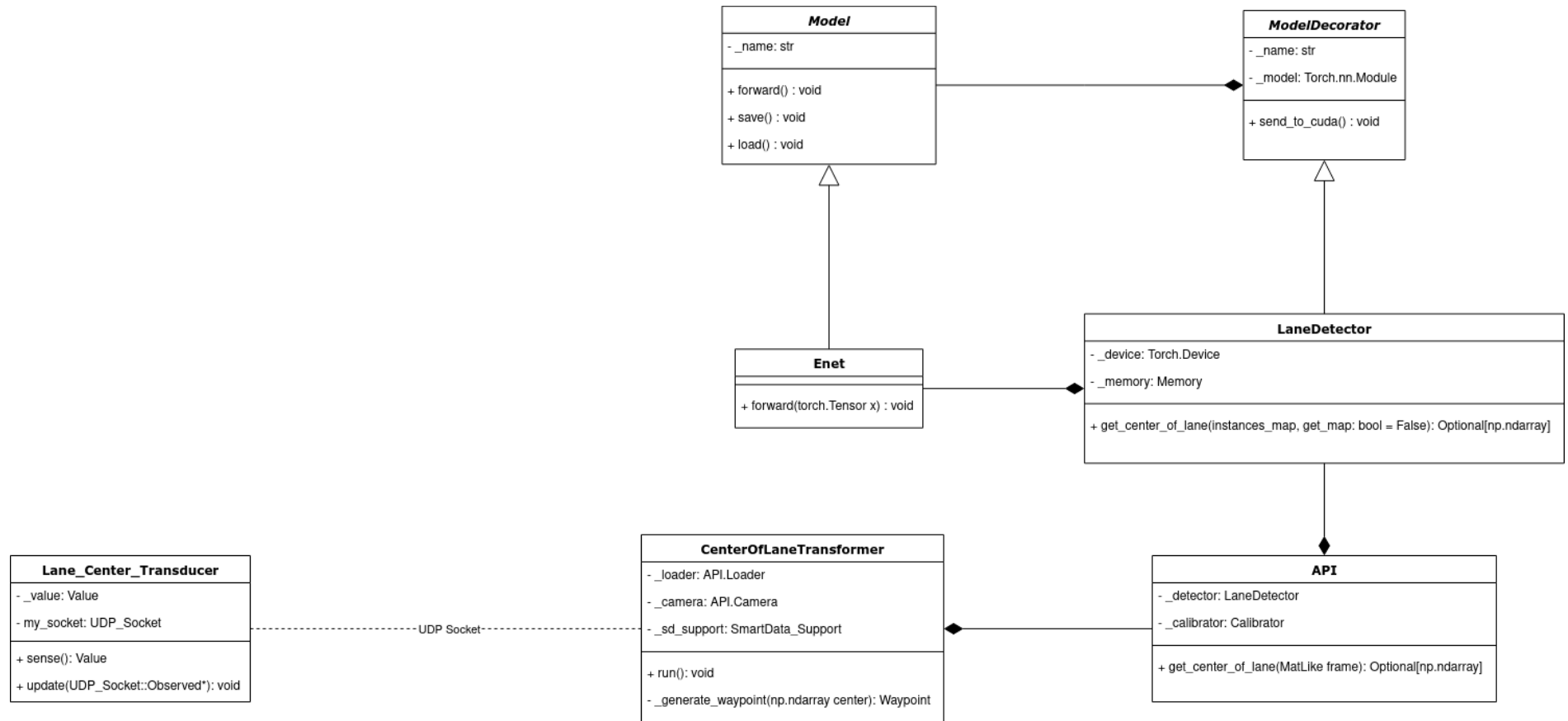
Function	Image segmentation to detect and track the navigable lane the vehicle is in.
FR	Average FPS of 30-33.
NFR	Real-time operation, max. energy xxx, max. GPU xxx, max CPU xxx.
Input	Raster images corresponding to individual camera frames represented as Image SmartData (UNIT = xx,xx,xx).
Output	Local Motion Vector SmartData (UNIT = xx,xx,xx) representing the center of the navigable lane.
Description	CNN based approach lane segmentation for center of lane extraction.
Code	GitLab .
Tests	Unitary , Integration .
Maintainer	Rodrigo Santos de Carvalho (git).

2. Strategy

The chosen approach basically consists of the implementation of the LaneNet also implemented by Neven et al in "Towards End-to End Lane Detection: an Instance Segmentation Approach".

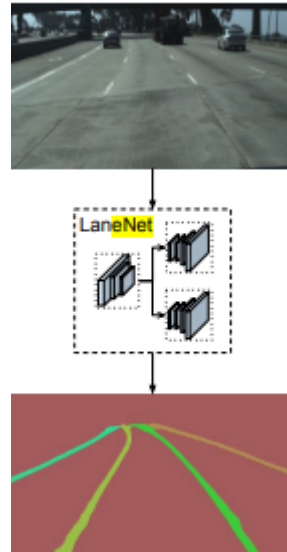
Python's Transformer send waypoints to C++'s SmartData Transducer via UDP Socket.

The following diagram explains how the created classes interact to provide the center of lane waypoint.



2.1. Steps

2.1.1. Segmentation



The network combines the benefits of binary lane segmentation with a clustering loss function designed for one-shot instance segmentation. In the output of LaneNet, each lane pixel is assigned the id of their corresponding lane. The instance segmentation task consists of two parts, a segmentation and a clustering part.

2.1.2. Path Constraining Lanes Extraction

After segmentation, to filter out the lanes that constrain vehicle's path, the distance between the center of the image and each segmented lane is calculated, and the closest lane on each side is considered to be the constraining lane on that corresponding side.

To avoid errors when segmenting other lanes in the fov of the vehicles camera, the distance between the found constraining lanes is calculated, and if its greater than a threshold (arbitrarily defined based on the position and kind of the camera), the frame is discarded.

2.1.3. Center of Lane in Meter Domain

With the filtered lanes, from the instance segmentation masks, the representative points of each lane is used to calculate the point of the center of lane. Since its all in the pixel domain, a conversion to meter domain is necessary to generate the center of lane Motion Vectors.

3. Algorithm

3.1. LaneNet

3.1.1. Binary segmentation

The segmentation branch of LaneNet is trained to output a binary segmentation map, indicating which pixels belong to a lane and which not, like a mask.

3.1.2. Instance Segmentation

To disentangle the lane pixels identified by the segmentation branch, we train the second branch of LaneNet for lane instance embedding.

By using De Brabandere et al clustering loss function, the instance embedding branch is trained to output an embedding for each lane pixel so that the distance between pixel embeddings belonging to the same lane is small, whereas the distance between pixel embeddings belonging to different lanes is maximized. So, the pixel embeddings of the same lane will cluster together, forming unique clusters per lane.

3.1.3. Network Architecture

Is based on the encoder-decoder network ENet. LaneNet only shares the first two stages (1 and 2) between the two branches. The last layer of the segmentation branch outputs a one channel image (binary segmentation), whereas the last layer of the embedding branch outputs a N-channel image, with N the embedding dimension. Each branch's loss term is equally weighted and back-propagated through the network.

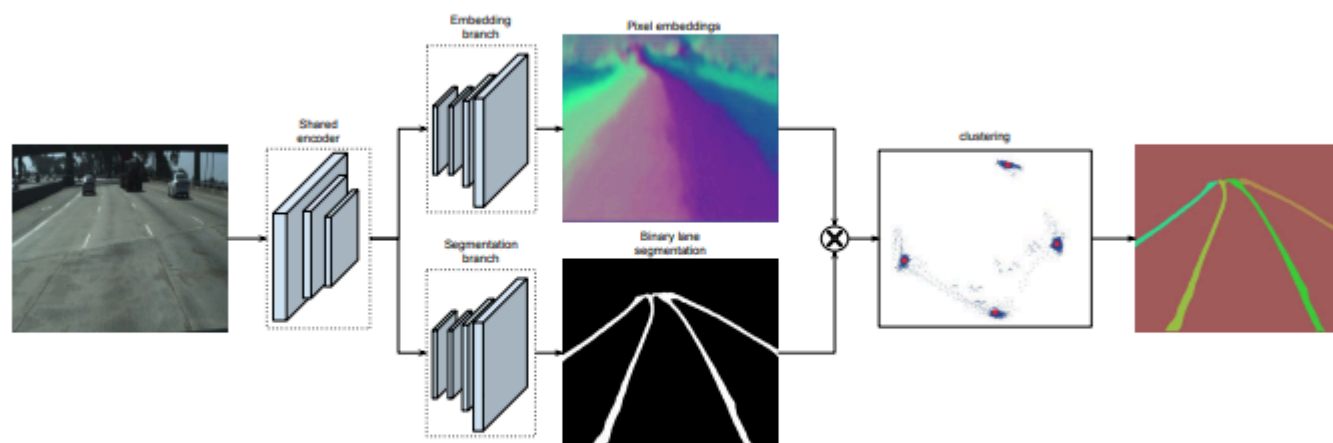


Fig. 2. LaneNet architecture. It consists of two branches. The segmentation branch (bottom) is trained to produce a binary lane mask. The embedding branch (top) generates an N-dimensional embedding per lane pixel, so that embeddings from the same lane are close together and those from different lanes are far in the manifold. For simplicity we show a 2-dimensional embedding per pixel, which is visualized both as a color map (all pixels) and as points (only lane pixels) in a xy grid. After masking out the background pixels using the binary segmentation map from the segmentation branch, the lane embeddings (blue dots) are clustered together and assigned to their cluster centers (red dots).

3.1.3.1. Forward



```
def forward(self, x):
    # Initial block
    input_size = x.size()
    x = self.initial_block(x)

    # Stage 1 share
    stage1_input_size = x.size()
    x, max_indices1_0 = self.downsample1_0(x)
    x = self.regular1_1(x)
    x = self.regular1_2(x)
    x = self.regular1_3(x)
    x = self.regular1_4(x)

    # Stage 2 share
    stage2_input_size = x.size()
    x, max_indices2_0 = self.downsample2_0(x)
    x = self.regular2_1(x)
    x = self.dilated2_2(x)
    x = self.asymmetric2_3(x)
    x = self.dilated2_4(x)
    x = self.regular2_5(x)
    x = self.dilated2_6(x)
    x = self.asymmetric2_7(x)
    x = self.dilated2_8(x)

    # stage 3 binary
    x_binary = self.regular_binary_3_0(x)
    x_binary = self.dilated_binary_3_1(x_binary)
    x_binary = self.asymmetric_binary_3_2(x_binary)
    x_binary = self.dilated_binary_3_3(x_binary)
    x_binary = self.regular_binary_3_4(x_binary)
    x_binary = self.dilated_binary_3_5(x_binary)
    x_binary = self.asymmetric_binary_3_6(x_binary)
    x_binary = self.dilated_binary_3_7(x_binary)
```



```
# stage 3 embedding
x_embedding = self.regular_embedding_3_0(x)
x_embedding = self.dilated_embedding_3_1(x_embedding)
x_embedding = self.asymmetric_embedding_3_2(x_embedding)
x_embedding = self.dilated_embedding_3_3(x_embedding)
x_embedding = self.regular_embedding_3_4(x_embedding)
x_embedding = self.dilated_embedding_3_5(x_embedding)
x_embedding = self.asymmetric_bembedding_3_6(x_embedding)
x_embedding = self.dilated_embedding_3_7(x_embedding)

# binary branch
x_binary = self.upsample_binary_4_0(x_binary, max_indices2_0, output_size=stage2_input_size)
x_binary = self.regular_binary_4_1(x_binary)
x_binary = self.regular_binary_4_2(x_binary)
x_binary = self.upsample_binary_5_0(x_binary, max_indices1_0, output_size=stage1_input_size)
x_binary = self.regular_binary_5_1(x_binary)
binary_final_logits = self.binary_transposed_conv(x_binary, output_size=input_size)

# embedding branch
x_embedding = self.upsample_embedding_4_0(x_embedding, max_indices2_0, output_size=stage2_input_size)
x_embedding = self.regular_embedding_4_1(x_embedding)
x_embedding = self.regular_embedding_4_2(x_embedding)
x_embedding = self.upsample_embedding_5_0(x_embedding, max_indices1_0, output_size=stage1_input_size)
x_embedding = self.regular_embedding_5_1(x_embedding)
instance_final_logits = self.embedding_transposed_conv(x_embedding, output_size=input_size)
```

3.1.3.2. Train

Params:

- Batch Size: 8
- LR: 0.0005
- Epochs: 20
- Adam Optimizer, weight decay: 0.0002



Discriminative Loss:

By using their clustering loss function, the instance embedding branch is trained to output an embedding for each lane pixel so that the distance between pixel embeddings belonging to the same lane is small, whereas the distance between pixel embeddings belonging to different lanes is maximized. By doing so, the pixel embeddings of the same lane will cluster together, forming unique clusters per lane.

```
def train(self, save=True):
    for epoch in range(self._NUM_EPOCHS):
        self._model.train()
        losses = []
        for batch in tqdm.tqdm(self._train_dataloader):
            img, binary_target, instance_target = batch
            img = img.to(self._device)
            binary_target = binary_target.to(self._device)
            instance_target = instance_target.to(self._device)

            self._optimizer.zero_grad()

            binary_logits, instance_emb = self._model(img)

            binary_loss, instance_loss = self._compute_loss(binary_logits, instance_emb, binary_target, instance_target)
            loss = binary_loss + instance_loss
            loss.backward()

            self._optimizer.step()

            losses.append((binary_loss.detach().cpu(), instance_loss.detach().cpu()))

        mean_losses = np.array(losses).mean(axis=0)

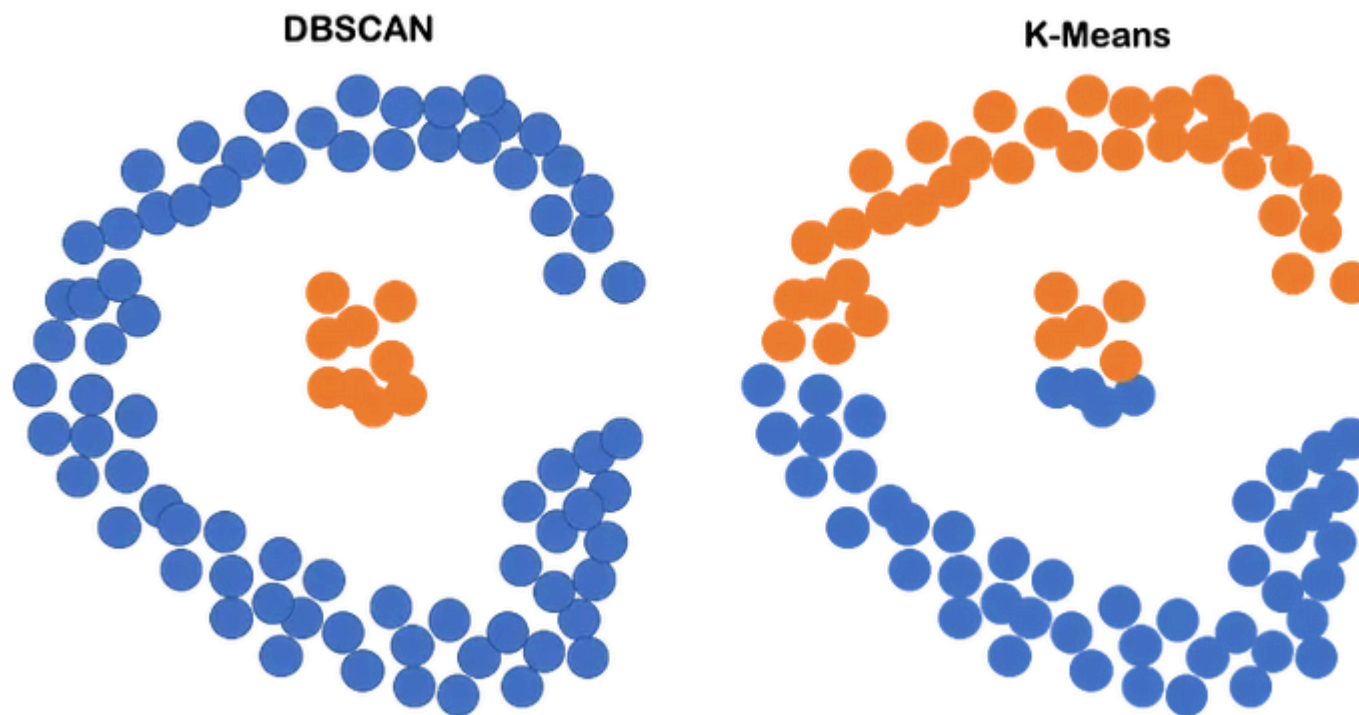
        msg = (f"Epoch {epoch}:"
              f"loss = {mean_losses}")
        print(msg)
```

```
if save:
```



3.2. DBSCAN

The clustering is done by an iterative procedure to select all embeddings belonging to the same lane. This is repeated until all lane embeddings are assigned to a lane. To avoid selecting an outlier to threshold around, we first use mean shift to shift closer to the cluster center and then do the thresholding.



3.3. Center Of Lane

The forward of the FNet network is clustered via DBSCAN. With that, an instance map is obtained. This map is like a mask where the values greater than 0 indicate lane-segmented points in the image. Thus, to get the separate lanes, the map is filtered. More lanes than the ones of interest may be detected, to eliminate those, the closest lane of each side of the center of the image is obtained. If the distance between the



found lanes is greater than a given threshold, they are discarded, as they probably contain a lane that is not the ones that constrains the car path. After finding the lanes of interest, the points outside of the bird's eye view area of transformation are masked out, to avoid errors in the next steps.



```
def _get_segmented_lanes(self, instances_map) -> Optional[Lanes]:
    filtered_instances = self._filter_instances(instances_map)

    # If there are no lanes or only one lane, return None
    # unless there are cached lanes
    if filtered_instances is None or len(filtered_instances) < 2:
        return self._memory.get_lanes()

    center_of_image = instances_map.shape[1] / 2

    lane_left_distances = []
    lane_right_distances = []
    for lane_mask in filtered_instances:
        # Get the lane position as the mean of the x axis
        lane_position = np.mean(np.where(lane_mask > 0)[1]) if np.any(lane_mask) else None

        if lane_position is not None:
            distance_from_center = center_of_image - lane_position

            if distance_from_center < 0: # Right lane, since the distance is negative, and the coords grow from left to
right
                lane_right_distances.append((lane_mask, distance_from_center))
            else: # Left lane, since the distance is positive, and the coords grow from left to right
                lane_left_distances.append((lane_mask, distance_from_center))

    if len(lane_left_distances) == 0 or len(lane_right_distances) == 0:
        return self._memory.get_lanes()

    # Sort the lanes by their distance from the center, getting the closest lane
    sorted_left_lanes = sorted(lane_left_distances, key=lambda x: x[1])
    sorted_right_lanes = sorted(lane_right_distances, key=lambda x: x[1], reverse=True)
```



```
# Get the masks of the closest lanes
left_mask, left_lane_distance = sorted_left_lanes[0]
right_mask, right_lane_distance = sorted_right_lanes[0]

# Check if the distance between the lanes is too big
max_distance_threshold = DecoratorConfig().max_distance
distance = left_lane_distance + abs(right_lane_distance)
if distance > max_distance_threshold:
    Logger.info(f"Distance between lanes is too big: {distance}")
    return self._memory.get_lanes()

left_mask, right_mask = self._crop_to_bev(left_mask, right_mask)
if self._not_enough_points_in_bev_region(left_mask, right_mask):
    return self._memory.get_lanes()

lanes = Lanes(left_mask, right_mask)
self._memory.update(lanes)
```

Now, with the lanes of interest, on each lane, the max point (x, y) is used to, by calculating the mean point between them, finally obtain the center of lane point. With that point, a calibrator object, which gets a camera as a parameter (which defines the intrinsic and extrinsic matrices), does the transformation from the image to real-world coordinates.

```
def get_center_of_lane(self, instances_map, get_map: bool = False) -> Optional[np.ndarray]:
    lanes = self._get_segmented_lanes(instances_map)
    if lanes is None:
        cached_lanes = self._memory.get_lanes()
        if cached_lanes is None:
            return None
        lanes = cached_lanes

    center_point = lanes.get_center_point
```





```
self._memory.update(center_point)

# Add center point mask
Logger.trace(f"Center point: {center_point}", show=True)

if get_map:
    instances_map = self._apply_center_mask(instances_map, center_point)

    return instances_map
else:
```

Note that the calibration must be done considering the resized 512x256 image that the model sees.

The following images illustrate the calculations below:





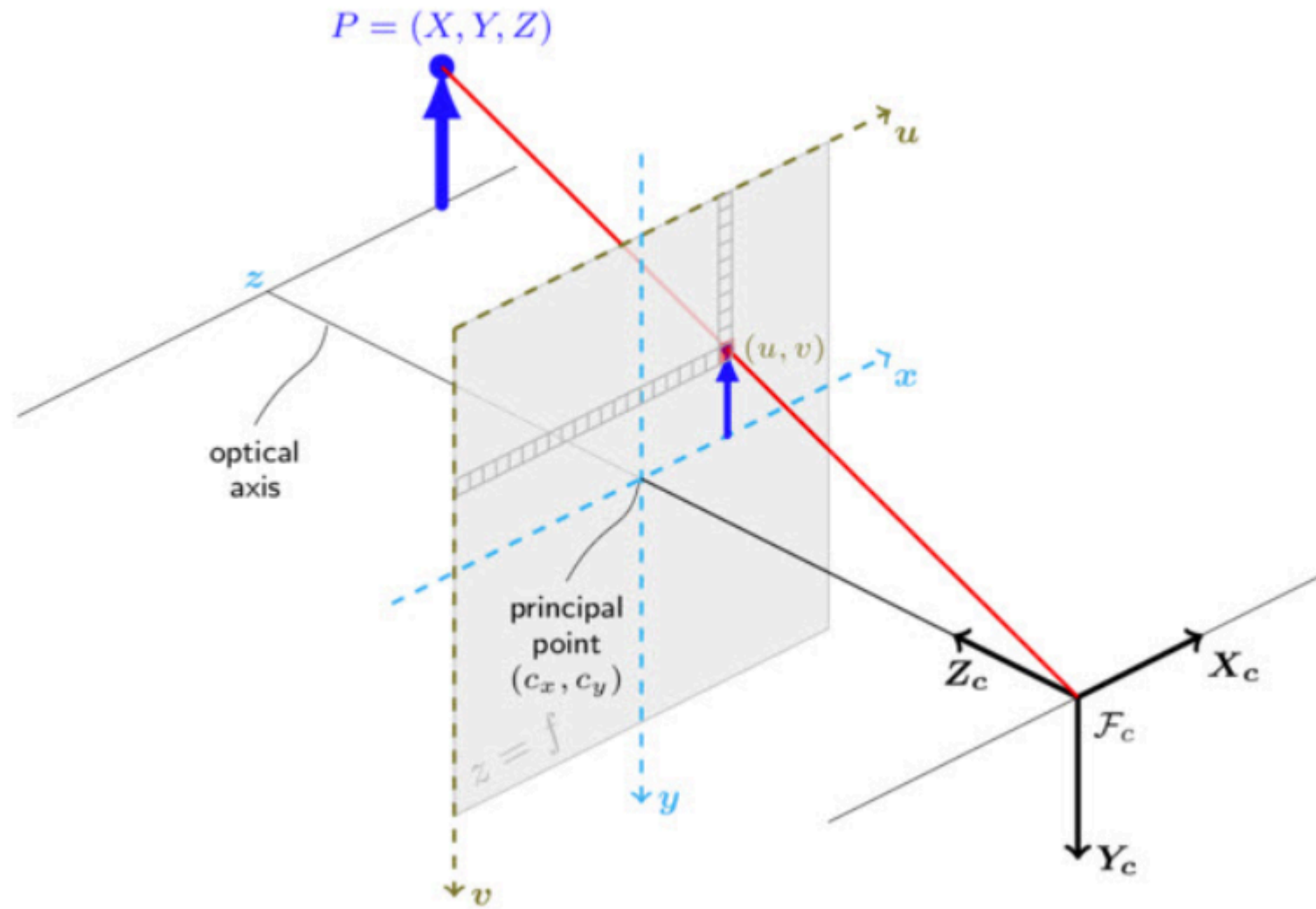
Given this
Find This

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

where:

- (X, Y, Z) are the coordinates of a 3D point in the world coordinate space
- (u, v) are the coordinates of the projection point in pixels
- A is a camera matrix, or a matrix of intrinsic parameters
- (c_x, c_y) is a principal point that is usually at the image center
- f_x, f_y are the focal lengths expressed in pixel units.

The pinhole camera model





The transformation above is equivalent
to the following (when $z \neq 0$):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

Pinhole model when z is not equal to 0

$$\left(s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} A^{-1} - t \right) R^{-1} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

solving for Real World XYZ coordinates

```
def to_real_world(self, pixel: np.ndarray) -> np.ndarray:
    """
    :param camera:
    :param pixel: pixel coordinate in the image
    :return: real world coordinate
    """
```





```

# S -> scale_factor
# I -> 3x3 intrinsic matrix
# E -> 3x4 extrinsic matrix # the rotation matrix that describes how the world coordinate system is rotated with
respect to the camera coordinate system. It defines the orientation of the camera in the world.
# P -> 3x1 image coordinates
# T -> 3x1 vector
# RWC -> 4x1 real world coordinates
S = self.camera.scale_factor
I = self.camera.intrinsics
E = self.camera.extrinsics
P = np.array([pixel[1], pixel[0], 1]) # in this context, [1] is x-axis and [0] is y-axis
T = self.camera.t # is the translation vector that describes the position of the world origin with respect to the
camera coordinate system. It specifies the location of the camera in the world


inv_I = np.linalg.inv(I)
inv_E = np.linalg.inv(E)

SP = S * P
SP_inv_I = SP @ inv_I
SP_inv_I_T = SP_inv_I - T
RWC = SP_inv_I_T @ inv_E

```

4. Easy Integration

To integrate and personalize the LaneDetection system, inside the CenterOfLaneTransformer, the loader and camera objects need to be changed.

It is easy to implement new Loader and Camera objects, as the interfaces are provided in the code. The Loader can retrieve images from the  the ones for the Carla simulator.



```
def run(self) -> None:
    print(self._NAME, "Run!")
    sync_time = float(sys.argv[1])
    time.sleep(sync_time - time.time())

    for frame in self._loader: # ADAPT HOW THE FRAMES ARE OBTAINED
        center = self._lane_detector.get_center_of_lane(frame)
        if center is None:
            continue

        waypoint = self._generate_waypoint(center)
        self._sd_support.write_data(waypoint)
        usleep(Data_Model.data_model[self._NAME][1])
        print("sent", waypoint)

    print("Finished!")
```

The following piece of code is where the Camera and Loader objects are passed to the Transformer.



```
if __name__ == "__main__":
    from data.loaders.video_loader import VideoLoader
    from camera.dummy_camera import DummyCamera

    loader = VideoLoader("lanedet.mp4")
    camera = DummyCamera()

    transformer = CenterOfLaneTransformer(camera, loader)
    index = 1

    transformer.run()
```

In C++, there's also a Center Of Lane transducer defined in transducer.h



```
#ifdef LANE_CENTER
#include "transducers/lane_center/lane_center.h"
#endif
```

5. API

An easy API is provided to use the LaneDetection system. There's also a config.json file that makes it simple to change some parameters of the code. The log can also be turned on, to aid in debugging if necessary.



```
class API:
    """
    Facade for the lane detection module.
    """
    Camera = Camera
    Loader = Loader

    def __init__(self, camera: Camera):
        self._detector = LaneDetector(ENet(2, 4))
        self._calibrator = Calibrator(camera)

    def get_center_of_lane(self, frame) -> Optional[np.ndarray]:
        """
        Returns the center of the lane as a numpy array,
        representing the point in the real world where the center of the lane is.
        :param frame:
        :return: The center of the lane as a numpy array. Can be None if no center of lane was detected.
        """
        instances_map = self._detector(frame)
        if instances_map is None:
            return None

        center = self._detector.get_center_of_lane(instances_map)
        if center is None:
```



```
        return None

    return self._calibrator.to_real_world(center)

def apply_center_segmentation(self, frame) -> Optional[cv2.typing.MatLike]:
    """
    Returns the frame with the center of the lane segmentation applied.
    :param frame:
    :return: The frame with the center of the lane segmentation applied. Can be None if no center of lane was detected.
    """
    instances_map = self._detector(frame)
    if instances_map is None:
        return None

    frame = cv2.resize(frame, LaneDetector.DEFAULT_IMAGE_SIZE)
    center = self._detector.get_center_of_lane(instances_map, get_map=True)
    if center is None:
        return None

    mask = self._make_mask(center)
    frame = self._apply_mask(frame, mask)

    frame = self._calibrator.bird_eye_view(frame)

    return frame

def apply_lanes_segmentation(self, frame) -> Optional[cv2.typing.MatLike]:
    """
    Returns the frame with the lanes segmentation applied.
    :param frame:
    :return: The frame with the lanes segmentation applied. Can be None if no lanes were detected.
    """
    instances_map = self._detector(frame)
    if instances_map is None:
        return None
```



```
instances_map = instances_map.cpu().numpy()

frame = cv2.resize(frame, LaneDetector.DEFAULT_IMAGE_SIZE)
mask = self._make_mask(instances_map)
frame = self._apply_mask(frame, mask)

frame = self._calibrator.bird_eye_view(frame)
```

6. Third-party References

[Towards End-to-End Lane Detection: an Instance Segmentation Approach](#)
[Coordinates: Bundler/VisualSFM, Matlab Calibration Toolbox, and OpenGL](#)
[DBSCAN](#)
[Calculate X, Y, Z Real World Coordinates from Image Coordinates using OpenCV](#)



Edit

Rename

Lock

History

Source

More