Perception and Decision Making in Intelligent Systems Homework 4

313554024 林慧旻

1. Code explanation

Please explain in detail how you implement the affine transformation, ego-motion calculation, the high-level concept of your control policy, and the process from the model's output to your final control.

Future Ego-Motion Calculation

```
    def get_future_egomotion(self, seq_x, seq_y, seq_theta):

        seq_theta_rad = np.deg2rad(seq_theta)
        future_egomotions = []

        for i in range(len(seq_x) - 1):
            dx = seq_x[i + 1] - seq_x[i]
            dy = seq_y[i + 1] - seq_y[i]
            dtheta =
            ((seq_theta_rad[i + 1] - seq_theta_rad[i]) + np.pi) % (2 * np.p

            cos_theta = np.cos(-seq_theta_rad[i])
            sin_theta = np.sin(-seq_theta_rad[i])
            ego_translation =
            np.array([cos_theta * dx - sin_theta * dy, sin_theta * dx + cos

            transformation_matrix = np.eye(4)
            transformation_matrix[:2, 3] = ego_translation
            transformation_matrix[:2, :2] =
            [[np.cos(dtheta), -np.sin(dtheta)], [np.sin(dtheta), np.cos(dth

            pose_vec =
            mat2pose_vec(torch.tensor(transformation_matrix, dtype=torch.fl
            future_egomotions.append(pose_vec)
```

```
        return torch.stack(future_egomotions)
```

- **Input Conversion**:
  - `seq_theta` (Here is yaw angles)are converted from degrees to radians for trigonometric computations.
- **Loop Through Consecutive Poses**:
  - For each pair of consecutive positions (`seq_x[i]`, `seq_y[i]`, `seq_theta[i]`), the relative translation (`dx`, `dy`) and change in orientation (`dtheta`) are computed.
- **Coordinate Transformation**:
  - `dtheta = ((seq_theta_rad[i + 1] - seq_theta_rad[i]) + np.pi) % (2 * np.pi) - np.pi`
  - This computes the relative change in orientation (dθ) between two consecutive yaw angles, ensuring that the result stays within the range[−π,π] radians. This range is commonly used to represent angles in 2D space.
- **Transformation Matrix Construction**:
  - `transformation_matrix[:2, :2] = [[np.cos(dtheta), -np.sin(dtheta)], [np.sin(dtheta), np.cos(dtheta)]]`
  - A 4×4 transformation matrix is built to represent the relative motion. It encodes both the relative translation and rotation between the two poses.
- **Pose Vector Conversion**:
  - The transformation matrix is converted into a pose vector `[dx, dy, dz, roll, pitch, yaw]` using the provided helper function `mat2pose_vec`. This format includes translation (in 3D) and rotation.
- **Result Aggregation**:
  - All pose vectors are collected into a list, which is then stacked into a tensor and returned.

Affine Transformation

Create Affine Transformation Matrix

```
    def create_affine_mat(self, x1, y1, theta1, x2, y2, theta2):

    # carla coordinate system ( meter --> pixel )
        bev_dim_x = int((32.0 - -32.0) / 0.25)
        bev_dim_y = int((32.0 - -32.0) / 0.25)

        matrix = None
        resolution = 0.25
```

```
        rad1 = np.deg2rad(theta1)
        rad2 = np.deg2rad(theta2)
        dtheta = rad2 - rad1
        rotate = np.array([
            [np.cos(- rad1), -np.sin(- rad1)],
            [np.sin(- rad1), np.cos(- rad1)]
        ])

        delta_pos = rotate @ np.array([x2 - x1, y2 - y1])
        dx_pixel = delta_pos[1]/resolution + bev_dim_x//2
        dy_pixel = -delta_pos[0]/resolution + bev_dim_y//2

        matrix = np.array([
            [np.cos(dtheta), -np.sin(dtheta), dx_pixel/(bev_dim_x /2)-1],
            [np.sin(dtheta), np.cos(dtheta), dy_pixel/(bev_dim_y /2)-1]
        ])

        return torch.tensor(matrix, dtype=torch.float32)
```

- **Convert Angles**:
  - `theta1` and `theta2` (yaw angles) are converted from degrees to radians using `np.deg2rad`.
- **Compute Rotation**:
  - A 2D rotation matrix (`rotate`) is created to rotate the translation vector (`x2 - x1, y2 - y1`) from world coordinates into the local ego vehicle coordinate system at `theta1`.
- **Compute Translation**:
  - The position change (`delta_pos`) is computed in the ego vehicle's coordinate system.
  - This is converted into pixel units (`dx_pixel`, `dy_pixel`) based on the resolution(0.25 here).
  - The pixel coordinates are adjusted to fit the BEV pixel space, ensuring that `(0, 0)` in world coordinates aligns with the center of the BEV space.
- **Affine Matrix Construction**:

$$\text{matrix} = \begin{bmatrix} \cos(\Delta\theta) & -\sin(\Delta\theta) & \text{dx\_pixel (normalized)} \\ \sin(\Delta\theta) & \cos(\Delta\theta) & \text{dy\_pixel (normalized)} \end{bmatrix}$$

  - The affine matrix encodes rotation (`dtheta`) and translation (`dx_pixel`, `dy_pixel`) in the BEV pixel coordinate space.

- The matrix format is compatible with OpenCV and PyTorch affine transformation functions.
- Translation terms are normalized to the range [-1, 1] by dividing by half the BEV dimensions.

- **Return Tensor**:
  - The resulting affine matrix is returned as a PyTorch tensor.

Warp Features

```
    def warp_features(self, x, affine_mats):

        grid = F.affine_grid(affine_mats, size=x.size(), align_corners=Fals
        warped_x = F.grid_sample(x, grid, mode='nearest', padding_mode='zer

        return warped_x
```

- **Generate a Sampling Grid:**
  - `F.affine_grid()` creates a grid of coordinates in the input feature map based on the affine transformation matrices:
    - `size=x.size()` ensures the output grid matches the shape of the input feature map.
    - `align_corners=False` ensures consistent interpolation behavior, especially at the edges of the grid.
- **Apply Affine Transformation**:
  - `F.grid_sample()` warps the input feature map x using the sampling grid:
    - `mode='nearest'` ensures that the nearest-neighbor interpolation is used to fill pixel values.
    - `padding_mode='zeros'` fills out-of-bound pixels with zeros.
    - `align_corners=False` ensures consistency with the grid generation.

Prediction

```
    def calculate_birds_eye_view_features(self, x, intrinsics, extrinsics,

        b, s, n, c, h, w = x.shape

        # Reshape
        x = pack_sequence_dim(x) # torch.Size([1, 3, 6, 3, 224, 400])->torc
        intrinsics = pack_sequence_dim(intrinsics)
```

```
        extrinsics = pack_sequence_dim(extrinsics)
        affine_mats = pack_sequence_dim(affine_mats)
        geometry = self.get_geometry(intrinsics, extrinsics)


        # TODO_3: Complete the calculate_birds_eye_view_features() function
        hidden, depth, cam_front = self.encoder_forward(x)
        x = hidden.clone()
        bev = self.projection_to_birds_eye_view(hidden, geometry)
        warped_features = self.warp_features(bev, affine_mats)
        warped_features = unpack_sequence_dim(warped_features, b, s)


        # Initialize temporally fused BEV feature tensor
        temporal_fused_features = warped_features.clone()  # Start with cur
        temporal_fused_features[:, 0] = warped_features[:, 0]
        for t in range(1, s):
            for i in range(t):
                temporal_fused_features[:, t] += (self.temporal_coef[0, t,
            temporal_fused_features[:, t] += warped_features[:, t]

        # ---------------
        geometry = unpack_sequence_dim(geometry, b, s)
        depth = unpack_sequence_dim(depth, b, s)
        cam_front = unpack_sequence_dim(cam_front, b, s)[:,-1] if cam_front
        return temporal_fused_features, depth, x, cam_front
```

- **Feature Encoding**:
  - The input features are passed through `encoder_forward()` to extract hidden features (`hidden`), depth information (`depth`), and an optional `cam_front` tensor.
- **Projection to BEV**:
  - The encoded features (`hidden`) are projected into the BEV space using `projection_to_birds_eye_view()`.
- **Ego-Coordinate Warping**:
  - The BEV features for all frames are warped into the ego vehicle's current coordinate frame using `warp_features()` with the precomputed `affine_mats`.
- **Temporal Fusion**:

$$\tilde{x}_t = b_t + \sum_{i=1}^{t-1} \alpha^i \times \tilde{x}_{t-i},$$

  - Temporal fusion combines BEV features across multiple time steps following the function in the paper, shown as above. A temporal discount factor `self.temporal_coef` is used. This factor weights past frame contributions, allowing the fused features to incorporate temporal context effectively:
    - At each time step `t`, sum the past frames up to `t-1` are scaled by powers of the temporal coefficients and added to the current warped features.

- **Unpacking and Final Adjustments and Return**:
  - The warped features and other outputs (`geometry`, `depth`, `cam_front`) are reshaped back to their original dimensions using `unpack_sequence_dim`. Then return `temporal_fused_features`, `depth`, `x`, `cam_front`

Planning

```
def get_our_control(output):

    steer = 0.0
    throttle = 1.0
    brake = 0.0

    obstacles = output.get('segmentation', None)

    if obstacles is not None and drivable_area is not None:
        obstacle_prob = torch.softmax(obstacles, dim=2)  # Shape: [1, 7,
        obstacle_map = obstacle_prob[0, 2, 1, :, :].cpu().numpy()  # Extr

        front_region = obstacle_map[80:100, 122:134]
        is_obstacle = np.max(front_region)
        is_obstacle_history.append(is_obstacle)

        if is_obstacle > 0.4:
            throttle = 0.0
            brake = 1.0
        else:
            throttle = 0.7
            brake = 0.0
```

```
    return carla.VehicleControl(steer=steer, throttle=throttle, brake=brake)
```

- **Extract Segmentation Probability Prediction Data**:
  - The `segmentation` output contains the predicted obstacle probabilities from `output.get('segmentation', None)`.
- **Compute Obstacle Probability Map**:
  - **Softmax Operation**: The segmentation output is converted into probabilities using `torch.softmax` along the segmentation classes.
  - **Extract Obstacle Map**: The obstacle class's probability map for the current time step is extracted(0:t-2, 1:t-1, 2:t.....) (at index `[0, 2, 1, :, :]`) and converted to a NumPy array for further processing.
- **Region of Interest**:
  - A **front region** of the map (`obstacle_map[80:100, 122:134]`) is selected. This region corresponds to a critical area selected directly in front of the vehicle.
- **Obstacle Detection**:
  - **Obstacle Probability Check**: `np.max(front_region)` determines the maximum probability of an obstacle in the ROI.
  - **Thresholding**: If the maximum probability exceeds `0.4`, an obstacle is considered present, and appropriate controls are applied:
    - **Throttle**: Set to `0.0` .
    - **Brake**: Set to `1.0`.
  - If no significant obstacle is detected, the vehicle continues forward:
    - **Throttle**: Set to `0.7`.
    - **Brake**: Set to `0.0` .

2. Results and discussion

a. Screenshot of the evaluation results on three scenarios.

Criterion	Result	Value
RouteCompletionTest	SUCCESS	100 %
OutsideRouteLanesTest	SUCCESS	0 %
CollisionTest	SUCCESS	0 times
RunningRedLightTest	SUCCESS	0 times
RunningStopTest	SUCCESS	0 times
MinSpeedTest	SUCCESS	252.3 %
InRouteTest	SUCCESS	
AgentBlockedTest	SUCCESS	
ScenarioTimeoutTest	SUCCESS	0 times
Timeout	SUCCESS	

Criterion	Result	Value
RouteCompletionTest	SUCCESS	100 %
OutsideRouteLanesTest	SUCCESS	0 %
CollisionTest	SUCCESS	0 times
RunningRedLightTest	SUCCESS	0 times
RunningStopTest	SUCCESS	0 times
MinSpeedTest	FAILURE	131.47 %
InRouteTest	SUCCESS	
AgentBlockedTest	SUCCESS	
ScenarioTimeoutTest	SUCCESS	0 times
Timeout	SUCCESS	

Criterion	Result	Value
RouteCompletionTest	SUCCESS	100 %
OutsideRouteLanesTest	SUCCESS	0 %
CollisionTest	SUCCESS	0 times
RunningRedLightTest	SUCCESS	0 times
RunningStopTest	SUCCESS	0 times
MinSpeedTest	SUCCESS	130.79 %
InRouteTest	SUCCESS	
AgentBlockedTest	SUCCESS	
Timeout	SUCCESS	

b. Discuss your results and observations.

- **Detection Error When Turning:**

  There is a right angle corner in scenario 1. When passing the corner, since the intereseted area is defined in front of the car, and distance is across opposite lane, so the cars in the opposiate direction will be detected, forcing the car to be freezed in the middle of the road.

- **Interested Region Selection:**

  If the Region is too large, then the car will be paused even when no other cars are around; on the other hand, if the region is too small, the safe driving distance will be too short to prevent the car from collision.

3. Question Answering

a. How do you think self-driving cars turn the surrounding images into BEV feature representations? What may be the advantages and disadvantages?

How do self-driving cars turn surrounding images into BEV feature representations?

1. I assum that the images from multiple cameras will be processed to extract features and predict depth. These features will then be lifted into 3D space using depth information, and aligned to a common coordinate system based on the ego-motion of the vehicle.
2. Following the step 1, features from past frames will be aligned and aggregated in 3D space to preserve geometric consistency and enhance robustness. Convolutional Network or some attention mechanisms will be utilized to refine features for object detection.
3. The 3D features are projected onto a BEV grid, which provides a top-down view of the surroundings of the target car

Advantages and Disadvantages

Advantages:

- **Perspective Issues**: BEV removes the distortion and occlusion problems associated with perspective camera views, providing a clearer spatial layout.
- **Easier to Control**: The top-down representation is clear for trajectory planning.
- **Integration of Multiple Features**: Aggregating features from multiple cameras and frames enhances robustness and the ability to perceive in dynamic sight.

Disadvantages:

- **Dependency on Accurate Depth Estimation**: Errors in depth prediction can significantly affect the accuracy of BEV representations.
- **Data Loss**: Projecting 3D features to a 2D plane can lead to the loss of height-related information, potentially impacting object recognition

---

b. End-to-end models refer to deep learning systems that map raw sensor data directly to the vehicle’s control outputs. While these models simplify the overall system by eliminating the need for manual integration of separate modules, they face significant challenges in terms of explainability. How do you think we can enhance the model’s explainability? If the model were more explainable, humans might trust autonomous vehicles more.

- Introduce some interpretable intermediate mechanism, such as semantic segmentation, object detection, or BEV representations.
- Use attention maps to highlight the areas in the input data that influence decisions, allowing humans to understand what the blacj box is focusing on at each stage of processing.
- Slice the model into several stages and set objectives for every stage. Then, visualize the model's outputs at each stage, in this homework such as showing predicted trajectories, detected obstacles, and traffic signs, which can be presented to the driver for better navigation.