Module 2: 3D Object Tracking & Motion Forecasting

1 Introduction

In this assignment, you will extend your knowledge of 3D perception by implementing a LiDAR-based 3D object tracking and motion forecasting system, similar to those used in real self-driving vehicles!

This assignment has two parts: object tracking and motion forecasting. As with the previous assignment, starter code is provided in the `csc490-projects` repository. We will continue to use Python and PyTorch. Instructions for how to install the required libraries and prepare the dataset are found in the repository. We have updated the `environment.yml` file to include additional packages for this assignment. Please run `conda env update --file environment.yml` to update your environment. **Note**: You should not introduce additional dependencies for this project.

Grading: This assignment will be graded out of 100 with up to 20 bonus points up to a total maximum of 120. One of either the Improved Tracker or the Improved Motion Forecasting parts is enough in order to get 100 points. You would need to explore both in order to potentially get to 120. The previous assignment's bonus points carry over to this one.

Late Policy: If you submit the assignment late, you will incur a 20 points penalty on this assignment. For each additional late day thereafter, you will incur an additional 10 points penalty.

Collaboration: For this assignment, you may work in groups of 1 to 2 people. In either case, the grading rubric will be the same. You may not reference or copy code from other sources. Please do your own work! We will be checking that indeed you coded the module.

Submission: When you finish the module and prior to the deadline, please submit the following files:

- `track_pred_writeup.pdf` – A write-up of your answers to questions in Parts A and B, which should have a high-level structure like:
  1. Object Tracking Questions
  2. Improved Object Tracking Report (if chosen)
  3. Motion Forecasting Questions
  4. Improved Motion Forecasting Report (if chosen)
- `track_pred_code.zip` – A copy of `csc490-projects`, with your changes from Parts A and B.

Improved Tracker and Motion Forecaster Grading We emphasize that you will not be graded on the performance of your tracking and motion forecasting. Instead, we are interested in the clarity of your report and presentation, and the quality of your analysis. Your report should be 2 to 3 pages in length and your presentation should be 5 minutes long (with an additional 5 minutes for Q&A).

As mentioned before, exploring either improved tracking or motion forecasting will be enough to have a maximum grade of 100. Exploring both increases the maximum grade to 120.

See below for the grading rubric for the **report** parts of the assignment.

1. **Improved Tracking/Prediction Report [30/20 points[1]]**
    - Motivation [20% of points]
        - What is the problem being tackled?
        - Why is this problem relevant?
        - Briefly (1 to 2 paragraphs) explain other approaches to this problem
        - Motivate why you chose your approach (i.e., the 2+ improved motion forecasting techniques you explored)
    - Techniques [40% of points]
        - Describe the intuition for why your approach would help
        - Articulate your approach mathematically / algorithmically, with an emphasis on the model (e.g., the new decoder if you are doing multi-future or probabilistic prediction) and on the loss function used to train it.
    - Evaluation [20% of points]
        - Articulate your evaluation plan mathematically / algorithmically
        - Show compelling qualitative results / comparisons
        - Show compelling quantitative results
        - Visualize at least one of the created methods
    - Limitations [20% of points]
        - Are there setups in which your approach fails?
        - Are there potential paths of future work to improve your approach?
2. **Presentation [10 points][2]**
    - Clarity of presentation visualizations, illustrations, etc)
    - Conciseness (motivate, describe and show results of your approach within the 5 minutes)
    - Preparedness (flow of presentation, answers to Q&A, etc)

2 Part A: Object Tracking [30 points]

In this part you will implement a multi-object tracker. Same as the previous assignment, starter code is provided in the `csc490-projects` repository. There, you will find a mostly complete codebase, with the caveat that some functions have been replaced with stubs which we have marked `# TODO: Replace this stub code.` In the upcoming questions, we will guide you through the process of implementing these functions. By the end of this section, you should have a fully functional object tracker!

[1]The first "Improved" exploration project will give you the first 30 points. Exploring the second direction will give you the remaining 20.

[2]The total points and time budget of the presentation is the same whether or not you go for the bonus points. If you do explore both "Improved" sections, the presentation should cover both of them but the actual bonus points will come from the report.

In the previous assignment, you implemented an object detector, which creates bounding boxes for vehicle objects in a single LiDAR sweep. In this module, you will implement a simple tracker that associates bounding boxes across a sequence of LiDAR sweeps. Specifically, given a sequence of detected bounding boxes for each LiDAR frame, our goal is to partition the bounding boxes into a set of tracklets, where each tracklet is associated with a unique object that moves across frames.

The tracker you will implement conducts tracking in an online manner. That is, we start with the first frame at time step 0, we then assign a unique object ID to every detected bounding box in the first frame, and initialize the respective tracklet to contain the associated bounding box. Then, at each time step t, we track the new detections by matching against previous observations. To simplify the process, instead of matching against all previous bounding boxes, we only associate the bounding boxes in the current frame with the bounding boxes in the previous frame at time step $t-1$. The implementation of this basic online tracker can be split into three parts:

1. **Two-Frame Tracking:** Given M detected bounding boxes from the previous frame and N detections from the current frame, compute an assignment matrix $\mathcal{A} \in \{0,1\}^{M \times N}$ where $\mathcal{A}(i,j) = 1$ indicates that bounding box $\mathcal{B}_i^{t-1}$ in the previous frame is matched to bounding box $\mathcal{B}_j^t$ in the current frame. Note that it is possible that for some i, $\mathcal{A}(i,j) = 0$ for all j, meaning that $\mathcal{B}_i^{t-1}$ has no match in the current frame and the associated tracklet has ended. On the other hand, if for some j, $\mathcal{A}(i,j) = 0$ for all i, it means that a new tracklet has emerged with $\mathcal{B}_j^t$ as its first detection.

2. **Multi-Frame Tracking:** Given a sequence of detections over multiple frames, we conduct online tracking by invoking the two-frame tracking function above. We initialize the tracklets with detections in the first frame, and use the assignment matrix $\mathcal{A}$ between consecutive frames to terminate or extend existing tracklets, or establish new tracklets.

3. **Evaluation:** Finally, to evaluate tracking performance, we compute the multiple object tracking precision (MOTP), multiple object tracking accuracy (MOTA), mostly tracked (MT), least tracked (LT) and partially tracked (PT) metrics.

In this module, you will be responsible for implementing functionalities in two-frame tracking and evaluation metrics. We will use the small subset of the Pandaset dataset [10] as in the previous assignment for experimentation. This dataset is split into a training set of 27 sequences and a validation set of 12 sequences. Each sequence consists of 80 LiDAR point clouds, each captured in intervals of 100ms. You will also find corresponding bounding box labels for each LiDAR point cloud. By default, we only conduct tracking on validation set. However, since our basic tracker does not involve any training procedures, we can also conduct evaluation on the training set, with the caveat that the results will not be comparable with learning-based tracking approaches.

As we need to track bounding boxes across the entire sequence, unlike object detection where we operate in the frame-wise vehicle coordinate frame, here we need to conduct tracking in a "global" coordinate frame. Specifically, we are using the world coordinate frame defined in Pandaset instead of the vehicle coordinate frame used in Assignment 1. Don't worry! We have already implemented the tracking dataset in `tracking/dataset.py` that prepares the tracking data and transforms the coordinates automatically. To ensure that there are no discrepancies in the detection inputs between different submissions, we have dumped the detections under `tracking/detection_results` as the inputs to the tracker. Please make sure you use the correct data (loaded by default). In other words, you do not need to generate and load the detection results by yourself.

In the upcoming questions, you will learn to implement a basic online tracker. Let's get started!

2.1 Two-Frame Tracking [15 points]

In this question, you will learn about two-frame tracking and implement member functions of the `Tracker` class in `tracking/tracker.py` and some related helper functions. The two-frame tracking process can be divided into three steps.

Part 1: Cost matrix construction. [3 points]

In this step, you will implement the `cost_matrix` function of the `Tracker` class. This function takes two sets of 2D bird's-eye-view (BEV) bounding box tensors $\mathcal{B}_1 \in \mathbb{R}^{M\times 5}$ and $\mathcal{B}_2 \in \mathbb{R}^{N\times 5}$ as input, where M and N are the numbers of bounding boxes present in the previous and current frame respectively, and each row $\mathcal{B}(i) = (x, y, l, w, \theta) \in \mathbb{R}^5$ stores the 2D position of the bounding box centroid, the length and width of the bounding box, and the heading angle of the bounding box. The goal is to output a matrix $\mathcal{C} \in \mathbb{R}^{M\times N}$ where $\mathcal{C}(i, j)$ is the cost/disaffinity score between $\mathcal{B}_1(i)$ and $\mathcal{B}_2(j)$.

To compute the disaffinity score between $\mathcal{B}_1(i)$ and $\mathcal{B}_2(j)$, we adopt a simple approach based on Intersection over Union (IoU). IoU is a metric that computes the overlap between two (rotated) bounding boxes. Figure 1 shows an illustration on how IoU is computed. You are responsible for implementing a helper function `iou_2d` in `tracking/cost.py` that computes the pairwise IoU value between bounding boxes in $\mathcal{B}_1$ and bounding boxes in $\mathcal{B}_2$. To compute the area of intersection or union between two bounding boxes, you can use the `Polygon` class from the `Shapely` package. Please see `https://shapely.readthedocs.io/en/stable/manual.html` for detailed documentation.

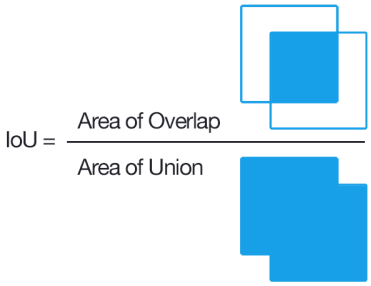

Figure 1: Illustration of 2D IoU.

To test the implementation of your `iou_2d` function, we have provided simple unit tests under `tracking/cost_test.py`. Under the `csc490` python environment, you can run them with `pytest tracking/cost_test.py`.

To implement the `cost_matrix` function of the `Tracker` class, you should first find the affinity/IoU scores by invoking the `iou_2d` function. However, to compute the final cost, since a smaller IoU corresponds to higher affinity between two bounding boxes, we set $\mathcal{C}(i, j) = 1 - \text{IoU}(\mathcal{B}_1(i), \mathcal{B}_2(j))$ as the cost/disaffinity score.

Part 2: Association. [10 points]

The association function takes two sets of 2D bird's-eye-view (BEV) bounding box tensors $\mathcal{B}_1 \in \mathbb{R}^{M\times 5}$ and $\mathcal{B}_2 \in \mathbb{R}^{N\times 5}$ and outputs an assignment matrix $\mathcal{A} \in \{0, 1\}^{M\times N}$ where $\mathcal{A}(i, j) = 1$ indicates that $\mathcal{B}_1(i)$ is matched to $\mathcal{B}_2(j)$. In this step, you will implement two different association algorithms, namely the `associate_greedy` function and the `associate_hungarian` function of the `Tracker` class.

To associate two sets of bounding boxes, we will first compute the cost matrix $\mathcal{C}$ as implemented in Part 1. Then, we aim to find an assignment matrix $\mathcal{A} \in \{0, 1\}^{M\times N}$ that minimizes the total cost $\sum_{(i,j)} \mathcal{A}(i, j)\mathcal{C}(i, j)$, with the constraints that (1) $\sum_j \mathcal{A}(i, j) \leq 1$ for all i, and $\sum_i \mathcal{A}(i, j) \leq 1$ for all j, and (2) $\text{rank}(\mathcal{A}) = \min(M, N)$, meaning that every bounding box has at most one association and there are no remaining bounding boxes from both sets yet to be assigned.

Greedy Association: With the computed cost matrix, the `associate_greedy` function makes a function call to the helper function `greedy_matching` in `tracking/matching.py`, which you will also implement. The greedy algorithm solves the assignment problem in $\min(M, N)$ iterations. It first initializes the set $S_1 = \{0, \ldots, M-1\}$ and $S_2 = \{0, \ldots, N-1\}$ to include all the candidates. Then, in each iteration, it finds the available $(i, j) \in S_1 \times S2$ pair that has the least cost $\mathcal{C}(i, j)$, makes them a matching pair, and removes i and j from S_1 and S_2 respectively. After $\min(M, N)$ iterations, at least one of S_1 and S_2 will be depleted, and we will have $\min(M, N)$ pairs of assignment. Your `associate_greedy` should then populate the assignment matrix $\mathcal{A}$ based on the matches.

However, the greedy algorithm does not always guarantee minimal total cost. Please provide an example cost matrix where the greedy algorithm does not lead to optimal total cost. Can you also come up with a scenario with two sets of bounding boxes where the resulting cost matrix does not lead to optimal assignment with the greedy algorithm? Please include the answer to these two questions in your write-up.

Hungarian Association: To properly solve the min-cost bipartite matching problem, the Kuhn–Munkres algorithm [5,8], or the Hungarian algorithm, is typically employed. Similar to the greedy algorithm, you will implement the helper function `hungarian_matching` in `tracking/matching.py`. To implement the Hungarian association, we do not require you to implement the Hungarian algorithm from scratch. Instead, you should use the `scipy.optimize.linear_sum_assignment` function, with detailed documentation in `https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.optimize.linear_sum_assignment.html`.

To test your implementations, we provide simple unit tests in `tracking/matching_test.py`, which you can run with `pytest tracking/matching_test.py`. Note that passing these tests does not guarantee that your implementation is correct. You should feel free to add your own unit tests.

Part 3: Post-processing. [2 points]

The association step will assign every possible $\mathcal{B}_1(i)$ (if $M \leq N$) or every possible $\mathcal{B}_2(j)$ (if $M \geq N$), but in reality it might be possible that certain detection has no match, in the case that the associated tracklet terminates early, or the associated tracklet starts in the middle of a sequence. In the post-processing step, you will complete the implementation of the `track_consecutive_frame` function of the `Tracker` class, where we filter out assignments with cost larger than a pre-defined threshold. Specifically, given the assignment matrix $\mathcal{A}$ derived from the association step above, we will set $\mathcal{A}(i, j) = 0$ if $\mathcal{C}(i, j) \geq$ self.match_th.

2.2 Multi-Frame Tracking

Given a sequence of detections over multiple frames, we first create unique object IDs for every bounding box in the first frame. Then, for each incoming frame, we conduct `track_consecutive_frame` and either terminate, extend or start a new tracklet based on the assignment matrix $\mathcal{A}$. We have implemented the function `track` of the `Tracker` class and there is no action item on your side. Feel free to check out the function implementation if interested.

2.3 Evaluation [15 points]

After implementing the `Tracker` class, we are able to load the provided detection bounding boxes and track objects for a given sequence! However, there is one final piece. In this question, you will implement the evaluation metrics. The starter code is in `tracking/metrics/matching.py`.

To evaluate the estimated tracklets against ground-truth tracklets, the first step is to associate estimated actors ("hypothesis") with ground-truth actors ("objects"). Simply put, at each time step t, we aim to associate every ground-truth object present in the current frame with a unique estimated hypothesis actor in the current frame. This correspondence finding process is rather non-trivial [1], and we have implemented it for you in the `establish_correspondences` function of the `Matching` class. The important thing to note is that at the end of this process, we will have a set of $M_t = \{o_i : h_j\}$ mappings in the current frame that matches object o_i with hypothesis h_j. If o_i is matched to some h_j in the current frame but is also matched to some h_k with $k \neq j$ in the previous frame, then this mapping counts as a mismatch. If a ground-truth object o_i is present in the scene but not matched to any hypothesis, it is considered as a miss. If a hypothesis actor h_j is present in the current frame but not matched to any ground-truth objects, it is considered a false positive.

The `establish_correspondences` function finds a mapping M_t between the ground-truth actor ID and the tracked actor ID in every frame, and computes key metrics such as number of matches, misses, false positives and mismatches at every time step. For each correspondence (o_i, h_j), we also store the matching distance defined as the bounding box IoU between object o_i and hypothesis h_j. Note that we will make use of the `iou_2d` function you implemented in `tracking/cost.py` to compute this distance, so make sure your implementation is correct. Now, it is your job to implement the following tracking metrics given these numbers.

Part 1: Multiple Object Tracking Precision (MOTP)

In this part, you will implement the MOTP metric [1] in `compute_motp` of the `Matching` class, defined as

$$\text{MOTP} = \frac{\sum_{i,t} d_t^i}{\sum_t c_t},$$

where d_t^i is the matching distance of each matched (object, hypothesis) pair at time step t, and c_t is the number of matches made at time step t.

Part 2: Multiple Object Tracking Accuracy (MOTA)

In this part, you will implement the MOTA metric [1] in `compute_mota` of the `Matching` class, defined as

$$\text{MOTA} = 1 - \frac{\sum_t m_t + fp_t + mme_t}{g_t},$$

where m_t, fp_t, mme_t and g_t are the number of misses, false positives, mismatches and ground-truth objects at each time step, respectively.

Part 3: Mostly Tracked (MT), Least Tracked (LT), Partially Tracked (PT)

We have provided the implementation of these three metrics for you in `compute_det_tracked_metrics` of the `Matching` class.

To test the implementation of your metrics function, we have provided simple unit tests. Under the `csc490` python environment, you can run `pytest tracking/metrics`.

Part 4: Evaluation and analysis

With the evaluation metrics implemented, now you are ready to run the full tracking pipeline! Run `tracking/main.py` to load the prepared detection bounding boxes, run the tracker and evaluate or visualize the resulting tracklets. You tracker should support both greedy and Hungarian association. In the following, we provide more instructions on how to run, evaluate and visualize our tracker:

- Load prepared detection bounding boxes, run the tracker and save tracking results in the pickle format: `python -m tracking.main track --dataset_path=$DATA_PATH --tracker_associate_method=hungarian` The results are saved to `tracking/tracking_results` by default. You can also change associate method to `greedy` and change the output path `--results_path` to save tracker results with greedy association algorithm. It usually takes several minutes to run on all 12 validation sequences. If you want to run the tracker on 27 training sequences, please specify the detection path as `--detection_path="tracking/detection_results/csc490_detector_train"`
- Evaluate the tracking results: `python -m tracking.main evaluate`

  You should specify `--results_path` if your results are saved at a different path in the previous step.
- Visualize the prediction and ground-truth tracklets: `python -m tracking.main visualize` The visualization creates two figures for each validation log: one for the ground-truth tracklets and one for the estimated tracklets. In each visualization, we associate each actor with a unique color and plot its bounding box centroids throughout the log in the Pandaset world coordinate frame.

In your write-up, include the final evaluation metric results for each of the 12 validation logs, as well as the median and mean results for all 12 validation logs. In other words, include the output of the `python -m tracking.main evaluate` in your report. How does the greedy association compare with the Hungarian association on our dataset?

2.4 Improved Tracker

The tracker that you have just implemented has lots of room for improvement. In this section you will have an opportunity to explore possible ways to improve tracking performance. Here are some potential directions you can explore:

- **More sophisticated cost functions:** In the basic tracker, we only leverage `1 - IoU` as a measure of the cost between bounding boxes $\mathcal{B}_1(i)$ and $\mathcal{B}_2(j)$. In practice, we might also need to consider other cost functions including (a) *geometry distance*: box size difference, centroid distance, heading angle distance; (b) *motion feature*: we can compute velocity based on existing tracklet and measure velocity difference; (c) *deep neural network feature*: distance between feature embeddings output by a neural network (*e.g.*, by your detection network from the last assignment).

  To explore this direction, please implement at least two different types of cost functions from the three categories above. Does adding or replacing with those additional cost functions help in our dataset? What are some other cost functions you can think of to improve tracking performance? Note that you might also need to tweak the `match_th` hyperparameter with different cost functions. You should tune your hyperparameters on the training set and evaluate on the validation set.
- **Use prediction to aid tracking:** Consider the case of two vehicles moving very fast one following another, it is possible for our tracker to associate them wrongly. If we can obtain the motion behavior or current velocity for these two actors, then we can predict the locations of actors in the next frame. After motion compensation, we are able to conduct tracking in a more static coordinate (actors with minor movements) thus leading to better performance. Can you leverage prediction results (from second part of this assignment) to aid tracking?

- **Occlusion handling:** How can we handle occluded or missing detections? That is, if one actor is occluded by other actors at some frame but shows up again in later frames or if our detector does not produce detection results for some actors at several frames, our current tracker will produce multiple tracklets for this single actor. Can you propose and implement ways to resolve this issue?
- **Learn to track:** Our basic tracker does not involve any learning. To learn tracking with ground-truth tracklets, a typical approach employs an LSTM network that aggregates features from previous frames, uses an MLP to produce an association score based on the previous frame features and the current frame detection features, and outputs an assignment based on the association scores (*e.g.*, with the Hungarian algorithm). We optimize the neural network parameters by minimizing the difference between estimated tracklets and ground-truth tracklets. In this way, we learn the features and costs used for association, instead of using *e.g.*, handcrafted geometric features. For this exploration direction, we ask you to explore learning-based tracking. This is a fairly complicated process and we do not expect you to finish the full implementation. If you would like to explore this direction, please include your full method formulation in your write-up including the network architecture, how the estimated tracklets are obtained, the training loss function based on ground-truth tracklets, as well as any code and intermediate results based on your proposed method. If you would like to learn more about how to jointly train detection, tracking and prediction, please see [7].

3 Part B: Motion Forecasting

Detection and tracking are concerned with building an understanding on what already happened in the world. In this part of the assignment, we will turn our attention towards the future, and build a model which attempts to predict where existing objects will go in the immediate future. Motion forecasting is used to help a motion planner determine where other actors will go in the future, and plan a safe route that avoids collision or unsafe buffer distances with them.

The relevant starter code is located in `prediction/`.

As we saw in class, there are countless ways of implementing a motion forecaster. Some are based on pure extrapolation (i.e., ballistic trajectory assumption) [6], some leverage the map heavily but without learning [13], some learn to perform non-linear extrapolation [3], while others jointly reason about detection, tracking, and prediction [7].

In this assignment, you will learn:

- How to use existing object tracks in order to make predictions about where each object will be at different times in the future,
- What the limitations of this first baseline implementation are, and how to identify and implement meaningful improvements, such as probabilistic motion forecasting, multi-hypothesis forecasting, interactive forecasting, etc.

To enable this kind of prediction, we provide the detections and trajectories produced by a known-good detector for the objects in the Pandaset. For each sample, an object's current and past detections will be used as inputs for the motion prediction.

Note that this part of the assignment does not rely on the outputs of your detector or tracker, or on any intermediate features produced by them.

The motion forecasting write-up should have two section:

1. the first one should contain answers to the questions encountered while progressing through the first part;
2. the second one should contain the description of the advanced motion forecasting implementation following the structure described at the end of the handout.

3.1 Baseline Motion Forecaster [30 points]

At each time step (t) there are N actor detections. The number of actors in a scene varies across time, as old actors disappear from view and new ones appear. For each detection, its current and past states need to be assembled as input to the MLP, according to the size of the time window being considered. We denote the window size as W. ($W = 10$ would consider the past 9 frames + the current one, thus 10, etc.) For each time step, we need to encode the object state at that time.

The baseline motion forecaster will predict the actor's future trajectory based on its past trajectory only (no LiDAR, no HD map, etc.). The inputs will be an object's current state together with its state history, and the outputs will represent the predicted state at future time steps $t_1, t_2, ..., t_T$, for a total of T steps, where T is a hyperparameter.

Prediction tasks are typically concerned with time horizons of 5–10 seconds, which would normally mean making predictions for 50–100 future time steps, assuming a 10Hz sensor rate, which is common in LiDAR-based autonomous driving.

In practice, it is not really necessary to predict waypoint coordinates for every single future time step, which can also be wasteful computationally. Instead, it is common to predict future waypoints at a lower frequency than the input frames, i.e., with a **stride**.

For the scope of this assignment, we will set $W = 10$. We predict 5 seconds into the future, at a stride of 5 frames. This means that we will predict the pose at $T = 10$ future positions: $t + 0.5s, t + 1.0s, \ldots t + 5.0s$.

Note that we do not stride the inputs to the network (the object pose history), just the outputs. Recall also that we will only have detection states as input, so there is no need to process LiDAR directly for motion forecasting.

Part 1: Encoding Past Trajectory Info (Prediction Encoder)

The first step is encoding existing trajectory information. This can be done for example with a multi-layer perceptron (MLP) architecture. For this simple baseline, we will use

$$(x, y) \in \mathbb{R}^2 \tag{1}$$

as features, where the x and y are expressed in vehicle frame at the current time t.

The input $\mathcal{X}$ to MLP will therefore have the shape: $\mathcal{X} \in \mathbb{R}^{W \cdot F}$, where F denotes the number of features per time step, so $F = 2$ in the example above. Note that in this case we want to consider each trajectory as its own sample, so we will perform MLP inference once for each trajectory, and output one feature vector for each trajectory.

The output of the encoder will simply take the form of a latent vector of neural net activations $f \in \mathbb{R}^{D_{\text{FEAT}}}$, where D_{FEAT} represents the dimensionality of the latent feature, and is a parameter that depends on the encoder architecture. In this assignment a value of $D_{\text{FEAT}} = 128$ is a good starting point. The code for the encoder should be put in the `PredictionModel` found in `prediction/model.py`.

Tip: The specific architecture of the MLP (e.g., the number of layers) is not very important. What's important is having a working system which produces plausible outputs (qualitative visualizations are important

here)—start with a small MLP and increase the size if needed while making sure you can iterate quickly enough.

Question: The code also contains an option to feed the current and past yaw as input to the model, in addition to x and y. Does adding yaw information help? Why / why not?

Part 2: Predicting Future Timesteps (Prediction Decoder)

The next step is designing how we wish to represent our predictions. We can make several assumptions about the future. One of them is that the object size will not change. This can help us define what our prediction neural network's output will be. Unlike a traditional ML problem like classification, we wish to predict multiple outputs. Specifically, for each of our T future time steps, we will want to predict the actor location in terms of x and y. While the yaw could also be a possible output, we will stick to just predicting x and y. If necessary, a good estimate of the yaw can simply be produced based on the predicted waypoint coordinates, but that is beyond the scope of this assignment.

For each of the N detections we will therefore want to output a tensor $\hat{\mathcal{Y}}_{\text{PRED}}^{(n)} \in \mathbb{R}^{N\times 2}$, where (n) represents the index of the actor.

As such, the decoder at this stage will also be an MLP which processes the encoder's latent vector, outputting the desired prediction tensor directly, i.e.,

$$\text{DEC} \ : \ \mathbb{R}^{D_{\text{FEAT}}} \rightarrow \hat{\mathcal{Y}}_{\text{PRED}}. \tag{2}$$

Put your code for the decoder in `PredictionModel` similarly found in `prediction/model.py`

Question: Right now our prediction has a similar output space to our detector - it predicts a box location at each future time. Can you think of other output parametrizations which can be used? What are their pros and cons?

Part 3: Implement the loss function in `prediction/modules/loss_function.py`, in the `forward` method of `PredictionLossFunction` and leverage it in the training loop from `main.py`.

The resulting neural network architecture is the composition of the decoder and the encoder. Mathematically, it has the following form:

$$\text{DEC}\left(\text{ENC}(\cdot)\right) \ : \ \mathcal{X} \rightarrow \hat{\mathcal{Y}}_{\text{PRED}}. \tag{3}$$

This network architecture can be trained using an ℓ_1 error between the predictions and the targets we just defined. Mathematically, for one frame of data at time t, this loss function is

$$\mathcal{L}\left(\mathcal{Y}_{\text{PRED}}, \hat{\mathcal{Y}}_{\text{PRED}}\right) = \sum_{n=0}^{N} \left\| \mathcal{Y}_{\text{PRED}}^{(n)} - \hat{\mathcal{Y}}_{\text{PRED}}^{(n)} \right\|_1. \tag{4}$$

Tip: You may be wondering what the point of separating the encoder from the decoder is, if they are both just simple multi-layer perceptrons. Surely, the result could also be obtained with just one big MLP mapping your inputs to the outputs, right? The answer to the second question it *yes*—we could just as well have used a single MLP with the layers of the encoder and the decoder concatenated together. However, as you will find in the later sections, especially in the "Improved Motion Forecaster" part, the decoder may not always be an MLP. Similarly, you may keep the decoder but modify your encoder, e.g., to process a prior map with a specialized architecture.

By conceptually separating the encoder and the decoder, it is easy to substitute different kinds of encoders or decoders while keeping the rest of the architecture unchanged.

Part 4: Overfitting Try overfitting to the predictions for the samples in a single frame of data. Include a visualization of the resulting trajectory forecasts alongside the ground truth in your report. To run the overfitting script after all the missing parts have been implemented, run the command

```
python -m prediction.main overfit --data_root=<path to pandaset dataset>
 --output_root=outputs
```

Part 5: Analysis Try varying two or three hyper-parameters, such as the encoder architecture, the amount of input frames used by the model, the trajectory input features, or the learning rate? What do you observe? What is the configuration which produces the best outcome on the validation set? To run the regular training loop, run the command

```
python -m prediction.main train --data_root=<path to pandaset dataset>
 --output_root=outputs
```

Perform your analysis in terms of the **average displacement error (ADE)** and **final displacement error (FDE)** metrics you saw in class, which are implemented in the assignment code base - see `prediction/metrics/ade_metrics.py`. After training you should be able to see the final results in the `outputs` folder. Make sure to run the command

```
python -m prediction.main evaluate --data_root=<path to pandaset dataset> --output_root=outputs
--checkpoint_path=outputs/<model_checkpoint>
```

Mathematically, ADE is averaged over all the N actors in a frame, and all T future time steps:

$$\mathrm{ADE}(\mathcal{Y}_{\mathrm{PRED}}, \hat{\mathcal{Y}}_{\mathrm{PRED}}) = \frac{1}{NT} \sum_{n=1}^{N} \sum_{t=1}^{T} \left\| \mathbf{y}_t^{(n)} - \hat{\mathbf{y}}_t^{(n)} \right\|_2^2. \tag{5}$$

Here, $\mathbf{y}$ can be seen as the t-th row of the n-th actor's prediction $\mathcal{Y}_{\mathrm{PRED}}^{(n)}$.

Similarly, FDE is computed for all N actors, but it only accounts for the offset at the end of each prediction:

$$\mathrm{FDE}(\mathcal{Y}_{\mathrm{PRED}}, \hat{\mathcal{Y}}_{\mathrm{PRED}}) = \frac{1}{N} \sum_{n=1}^{N} \left\| \mathbf{y}_T^{(n)} - \hat{\mathbf{y}}_T^{(n)} \right\|_2^2. \tag{6}$$

3.2 Improved Motion Forecaster

As we saw in the previous assignment, there are countless ways to improve the methods we just implemented. In this section you will analyze the existing motion forecaster's limitations, and propose and implement one or more potential improvements to it. Below are some examples, but you can propose other directions as well. If you do so, please check with a TA first.

Documenting and implementing ONE method is sufficient for getting the points for this part. Please refer to the assignment introduction for more information about grading and points.

- **Probabilistic Prediction:** So far, we made our prediction generate exactly one prediction for each actor as a point estimate. However, the future is uncertain, and beyond a very, very short future time horizon, there are multiple different things which can occur. In order to maximize safety, our SDV needs to be aware of this.

  One way of modeling this uncertainty is by producing probabilistic waypoints for the future predictions. In other words, instead of simply predicting an (x, y) location for each future waypoint, you can predict a probability distribution $p(x, y)$. The easiest way of representing this distribution is in a parametric way, using a Gaussian distribution. An example of such predictions is depicted in Fig. 2.

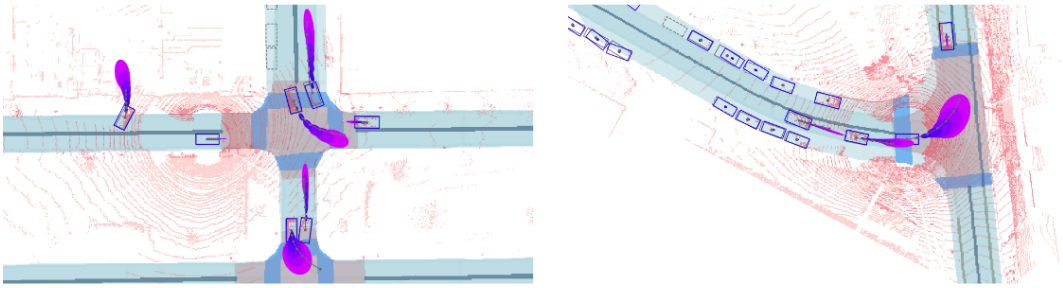

Figure 2: An example of predictions with Gaussian uncertainty. The gray trajectories represent the ground truth, while the blue ellipses correspond to one standard deviation. Illustration from Casas et al. [2].

The output would therefore need to replace the $(x, y) \in \mathbb{R}^2$ with $\mu = (x, y) \in \mathbb{R}^2, \Sigma \in \mathbb{R}^{2\times 2}$ for each waypoint, for each actor. Similarly, the loss function would need to be updated in order to enable learning probabilistic outputs. This can be done using a negative log-likelihood (NLL) loss, such as

$$\mathcal{L}_{\text{NLL}} = \sum_{t=1}^{T} \frac{1}{2} \log |\Sigma_t| + \frac{1}{2} (\mathbf{x}_t - \mu_t)^T \Sigma_t^{-1} (\mathbf{x}_t - \mu_t), \tag{7}$$

where $\mathbf{x}_t$ represents the current actor's ground truth future position at time t.

- **Multi-Modal Predictions:** Gaussian uncertainty can be powerful, but it is by its nature uni-modal. That is, it forces the network to predict a mean plus the covariance around it - a single "peak". However, this is not always accurate in the real world.

  Consider the uncertainty associated with whether a car in front of the SDV will change to a different lane vs. stay in its own lane. If the model believes there is a 50-50 chance it will do either of those things 5 seconds from now, how would it encode that?

  A Gaussian output would not be very informative - a net of this kind will likely smear the two equally likely outcomes together and predict a mean between the two possible futures. While this is not too useful for a SDV motion planner, it is the best that a unimodal Gaussian can do.

  One way to address this is to output multiple possible predictions for each actor. Specifically, instead of predicting T future positions for each actor (one per waypoint), the network would predict $K \times T$ positions—K possible futures. As before, we will need to adjust the basic predictor's loss function in order to accommodate this formulation.

  While in the past examples the network predicted one *thing* (i.e., one mode), whether it was a parametric distribution or a point estimate, now this is no longer the case. One of the main challenges with training such a motion forecaster is supervising the *multiple* alternatives it can compute. This is especially challenging because real-world training data only contains one future—the one which actually happened.

  In general, there are two steps that you should take to formulate the new loss function. Firstly, you should identify the best sample out of the predictions, and train it using the same ℓ_1 loss that was used previously. Next, you should train the model to predict which of the samples is the best one, so that it can automatically choose the mode without the guidance of the label.

- **Interactive Prediction:** So far, we have treated each actor's prediction as independent. However, as we saw in the lecture, one of the main challenges of motion forecasting is interaction modeling. In the real world traffic participants don't plan their trajectories in a vacuum. Instead, they plan, adjust,

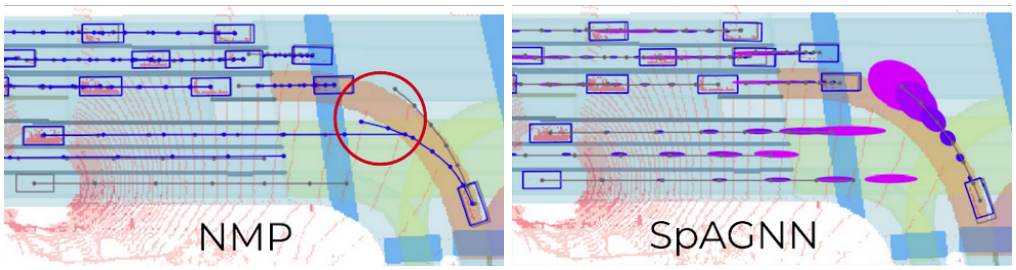

Figure 3: A comparison between predictions produced by the Neural Motion Planner [11], which does not enforce trajectory consistency, and those produced by SpAGNN [2], which does. Note how the predictions for the actor on the left collide in the NMP, but not for SpAGNN, which correctly models the fact that the actor on the left will slow down and yield to the one on the bottom-right. Illustration from [2].

and re-plan their immediate trajectories based on their surroundings, including their own intentions as well as their predictions for neighboring actors.

There is nothing stopping the past approaches from making infeasible predictions, like overlapping future actor trajectories, since predictions don't know anything about each other. An example of inconsistent vs. consistent trajectories is shown in Fig. 3.

One way to overcome this limitation is to treat the current outputs as initial guesses which can be refined by jointly reasoning about the entire scene.

Transformers [9] are one way to represent a neural network architecture which can attend to multiple elements (in our case, actor predictions), fuse information between them, and produce outdated outputs (consistent predictions). In PyTorch, one such module can be implemented based off the `torch.nn.MultiheadAttention` functionality, or using the built-in Transformer encoder (self-attention) directly - and instead of attending to words, the transformer would attend to trajectories.

- **Two-stage Prediction:** In motion forecasting, it is often useful to break down prediction into two parts: (1) predict where the end goal of the vehicle is (i.e. where it will be in 5 seconds), and (2) predict how the vehicle will get there (trajectory completion). The end goal has most uncertainty and variance of the waypoints in the future trajectory, so once you know where it is, it is often much easier to predict all the intermediate waypoints.

  Thus, you can develop a model to specifically predict the end goal, and then use a simpler model to predict the intermediate waypoints conditioned on the end goal. For example, recent works predict the end goal by predicting a dense heatmap of future end goals for each actor [4, 12]. However, we do not expect you to implement this - simpler implementation could predict the end goal directly, and possibly reweight the loss associated with the end goal vs. the other waypoints.

  Make sure to report how this implementation impacts the ADE, FDE, and model visualizations. See if you can explain why any changes occur.

References

[1] Keni Bernardin and Rainer Stiefelhagen. Evaluating multiple object tracking performance: The clear mot metrics. *J. Image Video Process.*, 2008, jan 2008. 6

[2] Sergio Casas, Cole Gulino, Renjie Liao, and Raquel Urtasun. SpAGNN: Spatially-aware graph neural networks for relational behavior forecasting from sensor data. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020. 12, 13

[3] Henggang Cui, Vladan Radosavljevic, Fang-Chieh Chou, Tsung-Han Lin, Thi Nguyen, Tzu-Kuo Huang, Jeff Schneider, and Nemanja Djuric. Multimodal trajectory predictions for autonomous driving using deep convolutional networks. In *2019 International Conference on Robotics and Automation (ICRA)*, 2019. 8

[4] Thomas Gilles, Stefano Sabatini, Dzmitry Tsishkou, Bogdan Stanciulescu, and Fabien Moutarde. HOME: heatmap output for future motion estimation. *CoRR*, abs/2105.10968, 2021. 13

[5] Harold W. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Research Logistics Quarterly*, 2(1–2):83–97, March 1955. 5

[6] Stéphanie Lefèvre, Dizan Vasquez, and Christian Laugier. A survey on motion prediction and risk assessment for intelligent vehicles. *ROBOMECH journal*, 1(1):1–14, 2014. 8

[7] Ming Liang, Bin Yang, Wenyuan Zeng, Yun Chen, Rui Hu, Sergio Casas, and Raquel Urtasun. Pnpnet: End-to-end perception and prediction with tracking in the loop. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11553–11562, 2020. 8

[8] James R. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, March 1957. 5

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 2017. 13

[10] Pengchuan Xiao, Zhenlei Shao, Steven Hao, Zishuo Zhang, Xiaolin Chai, Judy Jiao, Zesong Li, Jian Wu, Kai Sun, Kun Jiang, Yunlong Wang, and Diange Yang. Pandaset: Advanced sensor suite dataset for autonomous driving. *CoRR*, abs/2112.12610, 2021. 3

[11] Wenyuan Zeng, Wenjie Luo, Simon Suo, Abbas Sadat, Bin Yang, Sergio Casas, and Raquel Urtasun. End-to-end interpretable neural motion planner. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019. 13

[12] Hang Zhao, Jiyang Gao, Tian Lan, Chen Sun, Benjamin Sapp, Balakrishnan Varadarajan, Yue Shen, Yi Shen, Yuning Chai, Cordelia Schmid, Congcong Li, and Dragomir Anguelov. TNT: target-driven trajectory prediction. *CoRR*, abs/2008.08294, 2020. 13

[13] Julius Ziegler, Philipp Bender, Markus Schreiber, Henning Lategahn, Tobias Strauss, Christoph Stiller, Thao Dang, Uwe Franke, Nils Appenrodt, Christoph G Keller, et al. Making Bertha drive—an autonomous journey on a historic route. *IEEE Intelligent transportation systems magazine*, 6(2), 2014. 8