# Generating Trajectories: Route Planning Tools And Deep Learning Models

BENCHEKROUN Ali, BOUDARKA Omar
*École Polytechnique Fédérale de Lausanne*
14th June 2024

***Abstract*—The advancement of autonomous vehicle technology hinges on the ability to make intelligent decisions in complex traffic scenarios. This project focuses on developing a trajectory planning system that leverages predictive models to forecast the movements of other road users, thereby enhancing the safety, efficiency, and reliability of autonomous vehicles. Using the UniTraj Framework, we process diverse trajectory datasets to generate physically plausible paths for the ego vehicle. Our approach involves a multi-step pipeline that includes feature processing, path planning, model querying, and trajectory refinement. Key innovations include collision and boundary checking mechanisms that integrate into a cost function to evaluate and prioritize safe and optimal routes. Additionally, we implement the State Lattice Path Planner and a simple Physical Oracle approach to provide robust initial trajectory predictions. The final method incorporates a non-linear optimizer that refines the trajectory, ensuring compliance with dynamic constraints and driving comfort. This comprehensive framework aims to address the critical challenges in autonomous navigation, providing a reliable solution for real-world deployment.**

## I. Introduction

Autonomous vehicles must navigate complex traffic environments and make intelligent decisions to ensure safety and efficiency. The core challenge lies in accurately predicting the movements of other agents, including vehicles, pedestrians, and cyclists, to avoid collisions and maintain smooth traffic flow. Our project aims to address this challenge by developing a trajectory planning system that uses predictive models to forecast the behavior of surrounding agents. By integrating these predictions, the system can generate safe, efficient, and reliable paths for autonomous vehicles. Utilizing the UniTraj Framework, we process trajectory datasets to create realistic and feasible trajectories, incorporating state-of-the-art planning methods and optimization techniques to enhance autonomous navigation capabilities.

## II. Related Work

### A. *Game Former planner[1]*

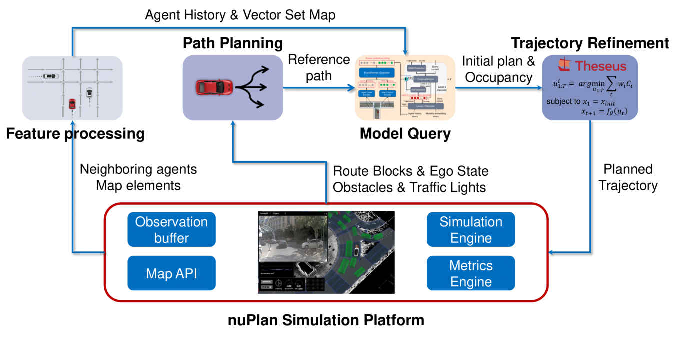

Fig. 1: U-Net 64 (32)

The GameFormer Planner is an advanced framework designed to enhance the decision-making capabilities of autonomous vehicles (AVs) by accurately predicting the future movements of surrounding agents and generating feasible and safe trajectories for the ego vehicle. This framework leverages hierarchical game theory and Transformer-based neural networks to handle complex and dynamic traffic scenarios. Here are the Core Components of the GameFormer Planner:

1) ***Feature Processing:***

*a) :* Input Data: The framework processes input features such as the historical trajectories of surrounding agents, the current state of the ego vehicle, and detailed map elements including lanes, crosswalks, and traffic signals.

2) ***Path Planning:***

*a)* ***Route Centerline:*** The path planning process begins by identifying the intended route for the ego vehicle, typically provided by a high-level route planner.

*b)* ***Candidate Paths:*** Using a state lattice approach, the planner samples multiple candidate paths along the route centerline. These paths are evaluated based on feasibility and alignment with traffic rules.

*c)* ***Optimal Path Selection:*** The paths are scored using a cost function that considers curvature, lane changes, obstacle avoidance, and other factors. The path with the lowest cost is selected as the initial plan.

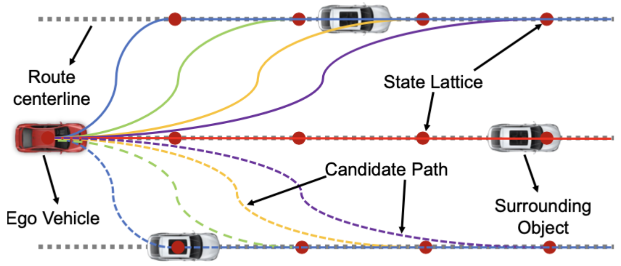

Fig. 2: Illustration of path generation process

3) ***Model Query:***

*a)* ***Initial Plan Generation:*** The GameFormer model generates an initial trajectory plan for the ego vehicle by encoding the scene context using a Transformer encoder. This context includes the historical states of all agents and local map information.

*b)* ***Trajectory Prediction for Surrounding Agents:*** The model predicts the future trajectories of surrounding agents through a hierarchical decoding process. This process iteratively refines the predictions by considering potential interactions between the ego vehicle and other agents.

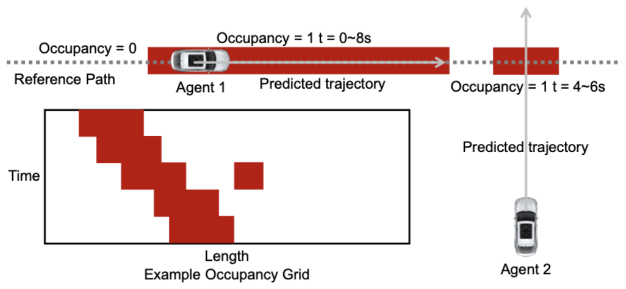

Fig. 3: Illustration of occupancy grid calculation

*4) **Occupancy Adapter**:*

*a) **Creation**:* The occupancy adapter predicts and manages the occupancy of a reference path by multiple vehicles over a series of time steps. It integrates vehicle trajectory predictions and their associated probabilities to map out the most probable future positions of vehicles. This helps in understanding which areas of the path are likely to be occupied.

*b) **Dynamic Update:**:* The occupancy grid is updated in real-time. The function identifies the most probable trajectory for each vehicle and converts these trajectories into a coordinate system relative to the reference path. If the reference path is too short, it is extended to accommodate all predicted time steps. A time occupancy matrix is created and updated to track the occupancy status of each segment of the reference path over time. For each time step, the function checks if a vehicle's predicted position falls within a safety threshold and marks the corresponding segments as occupied. This ensures that the occupancy status is accurately reflected, allowing the ego vehicle to navigate safely and avoid potential collisions.

*5) **Non-linear Optimization**:*

*a) **Refinement**:* The initial trajectory generated by the GameFormer model is refined through non-linear optimization to ensure comfort, safety, and compliance with traffic rules.

*b) **Cost Minimization**:* The optimization process minimizes a cost function that includes factors such as speed, acceleration, jerk (rate of change of acceleration), and distance to obstacles. This ensures the trajectory is not only feasible but also optimal for passenger comfort and safety.

## III. Final Method

### A. State Lattice Path Planner

*1) **Feature Processing**:* The State Lattice Path Planner leverages the UniTraj[2] framework to process various input features, including historical trajectories of vehicles and detailed map elements such as lanes, crosswalks, and road boundaries. These inputs are normalized relative to the ego vehicle's current position to ensure a coherent scene representation.

*a) **get_proximal_edges**:* The `get_proximal_edges`[0c] function identifies lane edges that are proximal to the ego vehicle's current position. Proximal edges are essential for forming the starting block for path planning, providing a set of possible initial states from which the path planning process can begin. This function calculates distances between the ego vehicle and lane segments to find the closest ones, categorized as either 'proximal_left' or 'proximal_right' based on their relative positions.

*b) **get_succesor_edges**:* The `get_succesor_edges`[0b] function identifies the successor edges for a given lane segment. Successor edges are important for extending the trajectory beyond the initial segment, allowing for a continuous and connected path. This function finds segments that follow the current segment in the driving direction, ensuring smooth transitions between lane segments.

*c) **find_closest_element_index**:* The `find_closest_element_index`[0d] function determines the current position of the ego vehicle relative to the map polylines. It identifies the closest segment on the map to the ego vehicle's current state by calculating the distance between the ego vehicle and each polyline segment and considering heading alignment. This ensures the vehicle is correctly aligned with the road, forming a reliable starting point for path planning.

*d) **Inputs to Planner**:* After processing the features and identifying the proximal and successor edges, the following inputs are fed into the State Lattice Path Planner:

- **Ego Vehicle State**: The current position, heading, and velocity of the ego vehicle.
- **Proximal Edges**: Lane segments that are close to the ego vehicle's current position.
- **Successor Edges**: Lane segments that follow the proximal edges in the driving direction.
- **Historical Trajectories**: Past positions and movements of surrounding vehicles.
- **Local Map Information**: Detailed map features, including lanes, intersections, and traffic signals.

*2) **Planner Process**:* The planner uses these inputs to generate feasible and optimal paths through the following steps:

*a) **Refining Starting Block**:* The starting block, consisting of proximal edges, is refined to include the most relevant and feasible lane segments based on the ego vehicle's current state and heading. This ensures a reliable starting point for the trajectory planning process.

*b) **Depth-First Search (DFS)**:* A depth-first search algorithm is used to explore all possible paths from the starting block. This exploration includes considering successor edges to determine viable routes that the ego vehicle can follow. The DFS ensures comprehensive path exploration by extending paths through connected lane segments.

*c) **Path Generation with Lattice and Bezier Curves**:* Paths are generated using a state lattice approach. Sample indices and Bezier curves are employed to create smooth and feasible trajectories that can handle various road geometries. This method ensures the paths are not only accurate but also smooth enough for practical driving.

*d) **Cost Function Evaluation**:* Each candidate path is evaluated using a detailed cost function that considers several factors:

- **Curvature**: Ensures the path is smooth and drivable.
- **Lane Changes**: Minimizes unnecessary lane changes to maintain safety.
- **Distance to Obstacles**: Avoids collisions with static and dynamic obstacles.
- **Traffic Rules Compliance**: Adheres to traffic regulations such as stopping at red lights and yielding.

The path with the lowest cost is selected as the initial plan.

*e) **Cubic Spline Interpolation**:* The selected path is refined using cubic spline interpolation, ensuring smooth transitions and continuity in the trajectory. This step enhances the comfort and drivability of the planned path by reducing abrupt changes in direction or speed.

By following these detailed steps, the State Lattice Path Planner generates optimal and feasible trajectories for the ego vehicle, ensuring safe and efficient navigation through complex and dynamic traffic environments.

## B. *Model Physics*

The primary goal of `physics.py` is to provide a framework for predicting vehicle trajectories using different physics-based methods. The model is designed to simulate vehicle movements based on velocity, acceleration, and other physical parameters, ensuring predictions are grounded in realistic motion principles.

*1) **Kinematics Extraction**:* Key kinematic data is extracted, such as position, velocity, acceleration, and yaw from the vehicle's trajectory records. This data provides a comprehensive snapshot of the vehicle's state at a given time.

*2) **Prediction Models**:*

*a) **Constant Velocity and Heading**:* Predicts future positions assuming the vehicle maintains its current speed and direction. This model is simple but effective for short-term predictions where the vehicle's behavior doesn't change abruptly.

*b) **Constant Acceleration and Heading**:* Extends the constant velocity model by incorporating acceleration. This model assumes the vehicle continues to accelerate at its current rate, providing a more dynamic prediction.

*c) **Constant Speed and Yaw Rate**:* Predicts the vehicle's trajectory assuming it maintains its current speed and yaw rate (rate of change of direction). This model is useful for scenarios where the vehicle is expected to follow a curved path.

*d) **Constant Magnitude Acceleration and Yaw Rate**:* Combines constant acceleration with a constant rate of directional change, offering a comprehensive prediction for vehicles making both linear and rotational movements.

*3) **Prediction Process**:* The model starts by extracting the vehicle's current kinematic state from the trajectory data. Using the extracted data, the model generates future trajectories based on different physics-based methods. Each method projects the vehicle's future positions over a specified time window. In the `PhysicsOracle` class, multiple trajectory predictions are evaluated against the ground truth data. The model calculates the error for each prediction and selects the one with the least deviation from the ground truth, ensuring the most accurate trajectory is chosen.

By leveraging these physics-based prediction models, `physics` provides a robust framework for simulating vehicle movements in various scenarios. This approach is particularly useful for short-term predictions and for vehicles operating under consistent motion patterns.

## IV. Metrics

### A. *Check Collision*

The `check_collision` function is a crucial metric used to evaluate the safety of the generated trajectories by detecting potential collisions between the ego vehicle and other vehicles. This function systematically checks for collisions over a series of time steps, ensuring that the planned trajectory avoids any conflicts with surrounding agents.

*a) **Collision Detection Process**:* The collision detection process involves several steps:

1) **Trajectory Extraction**: The function extracts the predicted trajectories of the ego vehicle and other vehicles for the specified time window.
2) **Bounding Box Calculation**: For each vehicle, the function calculates a bounding box that represents the vehicle's physical dimensions at each time step. This bounding box is typically a rectangle defined by the vehicle's length and width.
3) **Distance Calculation**: The function calculates the distance between the bounding boxes of the ego vehicle and each surrounding vehicle at each time step.
4) **Collision Check**: If the distance between any two bounding boxes is less than a predefined safety threshold, a potential collision is detected. The threshold is based on the combined dimensions of the vehicles plus a safety margin to account for uncertainties.

*b) **Safety Threshold**:* The safety threshold is a critical parameter in the collision detection process. It is determined by considering the physical dimensions of the vehicles involved and an additional safety margin. The safety margin ensures that even in the presence of minor prediction inaccuracies, the vehicles will not collide.

*c) **Output**:* The output of the `check_collision` function is a boolean value indicating whether a collision is detected. If a collision is detected at any time step, the function returns `True`; otherwise, it returns `False`. This information is used to refine the trajectory planning process, ensuring that the final trajectory is collision-free.

*d) **Implementation Considerations**:* When implementing the `check_collision` function, it is essential to consider:

- The accuracy of the predicted trajectories and bounding box calculations.
- The selection of an appropriate safety threshold to balance safety and feasibility.
- Efficient computation to allow real-time collision detection in dynamic environments.

By incorporating the `check_collision` metric, the trajectory planning system can ensure that the generated paths are safe and free from potential conflicts with other vehicles, enhancing the overall reliability of the autonomous driving system.

## V. Final Results

In our analysis, we are focusing on a qualitative assessment rather than a statistical one. This approach allows us to deeply understand the practical implications of the generated trajectories in various complex driving scenarios.

### A. *Complex Intersection*

In the first image, we observe a complex intersection. The State Lattice Path Planner (pink path) navigates smoothly through the intersection, maintaining a clear and feasible trajectory that takes into consideration the road's boundaries. Conversely, the Physics Model (yellow path) takes a more direct but less adaptive route, which may not effectively avoid potential obstacles or follow lane markings as accurately.

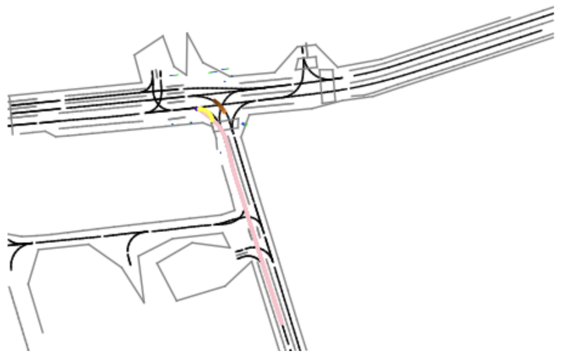
Fig. 4: Complex Intersection

### B. *Winding Road*

The second image showcases a winding road. Here, the State Lattice Path Planner again demonstrates its strength in handling curves. The Physics Model appears once again less adaptive, taking a more straightforward route that may not be as realistic for such winding trails.

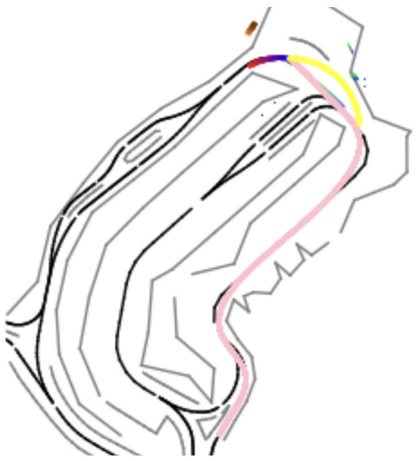

Fig. 5: Winding Road

### C. *Intricate Road Layout*

In the third image, we see another intricate road layout. The pink path shows a consistent ability to adapt to road changes. The yellow path takes a simpler, less flexible route that might not be practical in scenarios requiring precise navigation and obstacle avoidance.

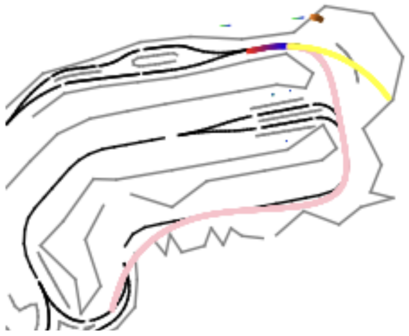

Fig. 6: Intricate Road Layout

**Overall Comparison**

Overall, the State Lattice Path Planner consistently produces more realistic and feasible trajectories, especially in complex road scenarios. In contrast, the Physics Model generates more direct but less adaptable routes, which may not always be suitable for intricate driving conditions. This qualitative analysis highlights the superior adaptability and practicality of the State Lattice Path Planner in managing real-world driving challenges.

## VI. Conclusion and Future Work

Our team has successfully implemented trajectory generation for the ego vehicle through various approaches. Moving forward, we plan to use models from the UniTraj framework to predict the behavior of surrounding agents. This will enhance our ability to generate accurate and realistic trajectories in dynamic environments.

*a) **Future Evaluation**:* For future evaluation, we will incorporate additional metrics such as miss rate and Prediction ADE (Average Displacement Error) into our `check_collision` function. These metrics will help ensure continuous improvement and reliability by providing more comprehensive insights into the performance of our trajectory generation models.

*b) **Outlook**:* The future looks promising as these advancements are built upon a robust foundation of research and development. By leveraging the UniTraj framework and refining our evaluation metrics, we aim to enhance the safety, accuracy, and efficiency of autonomous vehicle navigation in complex traffic scenarios.

Overall, our ongoing efforts will focus on integrating advanced prediction models, improving trajectory planning algorithms, and continuously validating our system's performance through rigorous testing and real-world validation.

## References

[1] X. M. C. L. Zhiyu Huang, Haochen Liu, "Gameformer planner: A learning-enabled interactive prediction and planning framework for autonomous vehicles,"

[2] K. M. B. A. Z. M. C. A. A. Lan Feng, Mohammadhossein Bahari, "Unitraj: A unified framework for scalable vehicle trajectory prediction," 2024.

## Appendix

*a) **INSERT_AGENT( ) FUNCTION:*** The `insert_agent` function is designed to place a new vehicle, referred to as an agent, onto a mapped area at a specific time. It ensures that the insertion adheres to spatial constraints to avoid collisions or overlaps with other vehicles. The inputs include:

- `obj_trajs`: a dataset containing the positions and other attributes of existing agents across the different time steps,
- `map_polylines`: information about the map (including road lanes and their characteristics) for the different time steps,
- `agent_info`: attributes of the new agent that need to be inserted (such as its dimensions and other relevant data),
- `point_coords`: the initial coordinates where the agent is intended to be placed, and
- `time_step`: the specific time step at which the agent should be inserted.

The function calculates a safety distance to ensure that the new agent will not be placed too close to any existing agent. This distance is based on the length of the new agent and the lengths of existing agents, adjusted by a safety factor to prevent overlaps. It then identifies the closest point on a lane to the specified insertion point. This involves measuring the distance from the specified coordinates to all points representing lane centers in the map and finding the nearest one and checks if there's already an agent at or near this closest lane point by calculating the distances from all existing agents at the specified time step to this point.

Afterward, it decides on the insertion of the new agent: if another agent is found within the calculated safe distance, the function returns the identifier of this nearest existing agent, indicating that the new agent cannot be inserted there without causing a conflict. If no existing agent is within this safe distance, the function proceeds to insert the new agent and updates the positions and attributes of all agents in the dataset to include the new agent's information, positioning it at the determined closest lane point. Finally, the function returns a tuple, either the dataset updated with the insertion of the new agent coupled to the coordinates of the insertion point if successful, or the input dataset coupled with the identifier of the conflicting agent if the insertion was not possible due to proximity issues.

*b) **GET_SUCCESSOR_EDGES( ) FUNCTION:*** The `get_successor_edges_bis` function is a static method designed to identify the successor and predecessor edges of specified lanes within a given map. This method processes the provided map polylines and their corresponding masks to establish the connectivity between different lanes based on spatial proximity. This is essential for understanding the flow of traffic and the layout of the road network.

The input parameters are:

- `map_polylines`: a dataset containing the polylines representing different lanes in the map, each polyline is an array of points that define the lane's shape and path;
- `masks`: a set of masks that indicate valid points within each lane's polyline, these masks help filter out irrelevant points; and
- `lane_idx`: a list of lane indices for which the successor and predecessor edges need to be identified.

The function first initializes an empty dictionary `suc` to store the successor and predecessor information for each specified lane. For each lane specified in `lane_idx`, the function extracts the mask and corresponding polyline points for the lane and initializes a dictionary `suc_info` to store the successor and predecessor lanes for the current lane. The function then iterates over all lanes in the map to determine their spatial relationship with the current lane. For each lane, it checks if the lane is not the same as the current one and extracts valid polyline points. It then calculates the distances between the endpoints of these lanes to see if they are close enough to be considered successors or predecessors. If the distance is below a certain threshold, it updates the current lane's successor and predecessor information accordingly. After that, the function updates the `suc` dictionary with the successor and predecessor information for that lane, and finally, the function returns the `suc` dictionary containing the successor and predecessor edges for each specified lane.

*c) **GET_PROXIMAL_EDGES( ) FUNCTION:*** The `get_proximal_edges_bis` function is a static method designed to identify lanes that are proximal to a given set of lanes within a specified distance threshold. This function processes map polylines and their corresponding masks to determine the lateral proximity of lanes, identifying those that are to the left or right of specified lanes.

The input parameters are:

- `map_polylines`: a dataset containing the polylines representing different lanes in the map, each polyline is an array of points that define the lane's shape and path;
- `map_polylines_mask`: a set of masks that indicate valid points within each lane's polyline, these masks help filter out irrelevant points;
- `successor_lane_ids`: a dictionary containing the successor and predecessor lanes for each lane;
- `lane_ids`: a list of lane indices for which the proximal lanes need to be identified; and
- `dist_thresh`: the distance threshold for considering lanes as proximal.

For each specified lane, the function extracts valid polyline points and initializes a dictionary to store information about proximal left and right lanes. It then iterates over all other lanes in the map, excluding lanes that are successors, predecessors, or the same as the current lane, to determine their proximity based on Euclidean distance calculations between the points of the current lane and the other lanes. If another lane is found to be within a specified distance threshold, the function calculates the orientation (yaw) of the lane segments at the closest points to determine the relative position, identifying if the lane is to the left or right of the current lane. This information is then stored in the dictionary. After processing all specified lanes, the function returns the dictionary containing the lateral proximity information for each lane, detailing which lanes are proximal to the left and right.

*d) **FIND_CLOSEST_ELEMENT_INDEX( ) FUNCTION:*** The `find_closest_segment_index` function is designed to identify the closest segment of a polyline from a set of map polylines relative to a given ego state. This method calculates distances between the ego vehicle's position and segments of the polylines, while considering certain conditions to determine the most relevant segment.

The input parameters are:

- `map_polylines`: a dataset containing the polylines representing different segments of the map, each polyline is an array of points that define the segment's shape and path; and
- `ego_state`: the current state of the ego vehicle, which includes its position and orientation.

The function begins by initializing variables to track the closest polyline and segment indices, with the initial minimum distance set to infinity. It then flattens the map polylines into a two-dimensional array for easier processing. For each polyline in this array, the function filters out zero points to focus on relevant segments. It calculates the Euclidean distances from the origin to each non-zero point and identifies the minimum non-zero distance among them. The function then checks if the current polyline segment meets the lane type conditions indicated in columns 9 to 13. If the segment's minimum distance is less than the previously recorded minimum and meets these conditions, the function updates the closest polyline and segment indices. Additionally, it ensures the heading alignment between the polyline segment and the ego vehicle by checking if their headings are close within a quarter of a circle. If so, it updates the minimum distance and considers this segment the closest. Finally, the function returns the indices of the closest polyline and segment.