Short-Horizon Path Planning for Urban Autonomous Driving Using a Conformal Lattice Planner

Khushal Brahmbhatt[1], Nicole Fronda[1], and Carson Gray[1]

***Abstract*— Vehicles with autonomous and semi-autonomous capabilities are becoming increasingly prevalent. A key component of the software stack for autonomous driving is motion planning. In this paper, we develop a solution for short-horizon planning using a Conformal Lattice Planner. We present a novel cost function for selecting the best trajectory generated by the Lattice Planner. We evaluate the performance of the Lattice Planner on map data and ground truth, human-driven trajectories provided by the Lyft Prediction dataset. We also compare the Lattice Planner trajectories and trajectories generated by a pre-trained Deep Neural Network (ResNet50) against human-driven trajectories.**

I. Introduction

Engineering robust and accurate motion planning solutions for autonomous driving on urban roads is central to making self-driving cars a reality. Although it remains unclear just how much self-driving cars are likely to improve road safety by [1], recent test data from key industry leaders [2] suggests they will have a significant impact in reducing the number of road accidents and fatalities per year.

An important part of motion planning in urban driving scenarios is short-horizon trajectory planning, which involves generating a feasible trajectory for the autonomous ego vehicle to follow over a short time period of a few seconds. This must take into consideration the kinematic and dynamic constraints on the vehicle, road geometry and traffic rules, as well as dynamic collision avoidance of surrounding obstacles. Complex driving scenarios and dense traffic make this problem inherently difficult.

In this paper, we use a conformal lattice planner on the Lyft Prediction dataset [3] to plan a collision-free path from the ego vehicle's current position to a goal state. By exploiting the structured nature of roads and only generating path options that swerve slightly to the left or right of the goal position, lattice planners generate smooth trajectories that closely resemble human driving [4]. Generating a collision-free lattice structure requires state input of the ego vehicle as well as predicted trajectories of surrounding vehicles at each timestep. Motion planning in autonomous driving also typically uses a receding horizon approach, which requires replanning at each step or every few steps along the planned trajectory, thus making this a sequential decision making problem. We use vehicle states and encoded maps from the Lyft Prediction dataset to set the kinematic and geometric constraints for generating the lattice.

[1]Collaborative Robotics and Intelligent Systems (CoRIS) Institute, School of Mechanical, Industrial, and Manufacturing Engineering, Oregon State University at Corvallis, OR 97331, USA. Email: {brahmbhk, frondan, graycar}@oregonstate.edu

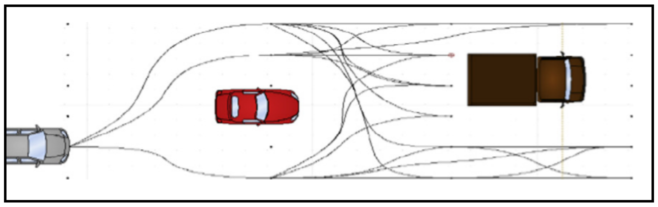


Fig. 1: An example of a lattice planner for short-horizon autonomous vehicle path planning [5].

Our main contributions in this paper are the following:

1) A conformal lattice planner to plan short-horizon trajectories on the Lyft Prediction dataset using unique cost functions.
2) A quantitative and qualitative comparison of our method against the existing Lyft baseline that uses a deep learning approach.

The rest of the paper is organized as follows: section II discusses previous works done in the field in terms of learning-based and classical approaches. Section III outlines the problem formulation. Our experimental setup is presented in IV and the results are presented and discussed in V. Finally, we present our conclusions and future scope in VI.

II. Related Work

Recent works have tried to sidestep most of the difficulties presented in Section I by directly learning from semantic maps using deep learning methods. Houston et al. [3] use a Convolutional Neural Network (CNN) to directly predict trajectories of surrounding vehicles from semantic map images generated from the dataset containing road geometry as well as obstacle masks and their history states. This method can be extended to plan the trajectory of the ego vehicle [6]. By combining CNNs with Long Short-Term Memory (LSTM) networks, [7] show that history state information of the vehicle such as its speed, acceleration and heading can also be leveraged along with the semantic maps to predict more accurate trajectories. The authors of [8] use a single neural network to jointly perform object detection, motion prediction and motion planning. Most deep learning-based motion prediction methods can also be extended to do motion planning. Thus an in-depth review of deep learning methods for behavior prediction for autonomous driving applications can be found in [9].

While deep learning solutions alleviate some of the difficulties associated with motion planning in urban driving scenarios, they lead to unexplainable solutions that are difficult to evaluate during failure cases, thus making it difficult to justify the safety of the autonomous driving system. State lattice planners [10] use real-world physics constraints to generate trajectories that also more closely resemble road driving patterns, shown in Figure 1. The authors of [4] added time and velocity dimensions to traditional lattice planners to create spatiotemporal planners capable of handling dynamic road environments. The authors of [11] built on this approach by making it more efficient and adding more realistic vehicle kinematics, and their approach is followed in this paper. A survey of non-deep learning methods for autonomous vehicle motion planning can be found in [12], [13].

III. Methods

A. Lyft Prediction Dataset

Our conformal lattice planner is implemented on the Lyft Prediction dataset [3]. The dataset contains over 1,000 hours of driving data along the same 6.8 mile stretch in Palo Alto over a 4 month time period. It is made up of 170,000 25-second scenes that include a detailed semantic map of the area with 15,242 labelled elements, an aerial camera view, and the perception output from the vehicles. The dataset was released in 2020 to accelerate autonomous driving research.

B. Lattice Path Generation

The goal of the conformal lattice planner is to find a kinematically feasible, obstacle-free path to a short term goal. The planner takes advantage of the structured nature of roads to speed up computation time. For example, it assumes that the ego vehicle is moving forward along the path of a known, mapped road. The road's lane boundaries are used to limit the search space. Overall, the lattice planner creates smooth plans that resemble those taken by human drivers [4].

The planner starts by choosing an initial goal state centered within the proper lane boundaries. It then generates paths that are laterally offset from the the initial central goal. When deciding the initial goal state, there is a tradeoff between informativeness and computation time [11]. Shorter paths require less computation, but do not take into consideration obstacles farther along the road. This tradeoff is especially important to consider in terms of the ego vehicle's velocity and the road conditions.

After determining goal states, the next step is to calculate a path to those states. Our method follows the approach of [11] and [14]. At this stage of the planner, it does not matter whether the paths overlap with obstacles, only whether the path is kinematically feasible. In autonomous driving, both quintic splines [15] and cubic spirals [16] have been successfully used for trajectory planning, parameterized for vehicle position and heading. To promote maximum comfort for vehicle passengers, we minimized the curvature of the goal path to allow for gradual navigation without sharp turns. Because of our curvature constraints, we chose to use cubic spirals, which do not change as quickly as the quintic splines. This allows us to constrain the entire curve by constraining only a handful of points [14].

Shown in Eq. 1, our curvature k is a function of arc length s. We can evaluate the vehicle heading Θ at any point s along the curve using Eq. 2. To find vehicle position (x, y), we numerically evaluated the Fresnel integrals in Eq. 3 using $\Theta(s)$ and the trapezoid rule [14].

$$k(s) = a_3 s^3 + a_2 s^2 + a_1 s + a_0 \tag{1}$$

$$\begin{aligned} \Theta(s) &= \Theta_0 + \int_0^s a_3 s'^3 + a_2 s'^2 + a_1 s' + a_0 ds' \\ &= \Theta_0 + a_3 \frac{s^4}{4} + a_2 \frac{s^3}{3} + a_1 \frac{s^2}{2} + a_0 s \end{aligned} \tag{2}$$

$$\begin{aligned} x_s(s) &= x_0 + \int_0^s cos(\Theta(s'))ds' \\ y_s(s) &= y_0 + \int_0^s sin(\Theta(s'))ds' \end{aligned} \tag{3}$$

The next step is to find the optimal cubic spiral from starting point $(x_0, y_0, \Theta_0, k_0)$ to goal point $(x_f, y_f, \Theta_f, k_f)$, where the curvature is minimized to account for the vehicle turning radius and passenger comfort. The objective function we minimize is the bending energy of the curve, shown in Eq. 4. By minimizing this function, we distribute the absolute curvature as evenly as possible along the path [14].

$$\begin{gathered} min f_{be}(a_0, a_1, a_2, a_3, s_f) = \\ \int_0^{s_f} (a_3 s^3 + a_2 s^2 + a_1 s + a_0)^2 ds \end{gathered} \tag{4}$$

To account for curvature, we constrained the curvature at two intermediate points, $k(s_f/2)$ and $k(3s_f/2)$, spaced one third and two thirds along the path. We set these against a maximum value, $k_{max} = 0.5$, which corresponds to a turning radius of 2m. We also added soft constraints for x, y, and Θ to make sure that a feasible path from start to end could be found within the constraints, scaled by α, β, and γ. The optimization problem is shown in Eq. 5.

$$\begin{aligned} p = min f_{be}&(a_0, a_1, a_2, a_3, s_f) \\ &+ \alpha(x_s(s_f) - x_f) \\ &+ \beta(y_s(s_f) - y_f) \\ &+ \gamma(\Theta(s_f) - \Theta_f) \end{aligned} \tag{5}$$

$$\text{s.t.} \begin{cases} \left|k(s_f/2)\right| \leq k_{max} \\ \left|k(3s_f/2)\right| \leq k_{max} \\ k(s_f) = k_f \end{cases}$$

To reduce the dimensionality of the problem, we can take advantage of the fact that we know the starting and ending conditions of the path. We can define curvature parameters $p_0 = k_0$, $p_1 = k(s_f/2)$, $p_2 = k(3s_f/2)$, and $p_3 = k(s_f)$ as four evenly spaced values along the curve, with $p_4 = s_f$ representing the arc length. There is a convenient closed form mapping between the curvature parameters and the spiral parameters [14], shown in Eq. 6.

$$
\begin{aligned}
a_0 &= p_0 = k_0 \\
a_1 &= -\frac{11p_0/2 - 9p_1 + 9p_2/2 - p_3}{p_4} \\
a_2 &= \frac{9p_0 - 45p_1/2 + 18p_2 - 9p_3/2}{p_4^2} \\
a_3 &= -\frac{9p_0/2 - 27p_1/2 + 27p_2/2 - 9p_3/2}{p_4^3}
\end{aligned}
\tag{6}
$$

Because p_0 and p_3 are the known start and end values of the curve, the new optimization variables are p_1, p_2, and p_4. Once again, these represent the curvature values one third and two thirds along the curve, respectively, as well as the final arc length s_f. The final optimization problem is shown in Eq. 7. Note that a_0, a_1, a_2, and a_3 should be represented by the parameter values in Eq. 6 throughout the calculation. The constraints in Eq. 7 represent the two intermediate points along the curve whose curvatures we constrain.

$$
\begin{aligned}
p = min f_{be}&(a_0, a_1, a_2, a_3, p_4) \\
&+ \alpha(x_s(p_4) - x_f) \\
&+ \beta(y_s(p_4) - y_f) \\
&+ \gamma(\Theta(p_4) - \Theta_f)
\end{aligned}
\tag{7}
$$

$$
\text{s.t.} \begin{cases} |p_1| \le k_{max} \\ |p_2| \le k_{max} \end{cases} \tag{8}
$$

Once the parameters have been found, they can be mapped back to the spiral coefficients using Eq. 6. The curve can then be sampled using $\Theta(s)$, x_s, and y_s (Eqs. 2 and 3).

C. Lattice Search

We consider the state of the vehicle as its (x,y) position and heading. Given the start state x_0 and goal state x_g at a particular point in time, we generate n sample trajectories using the **getSamples** procedure. This procedure first sets n different goal position offsets from x_g, and then generates paths to each of these goal offsets by solving the optimization problem (7)-(8). Finally, it uses the solved spiral coefficients to generate points between the initial state x_0 and each goal offset, as described in Section III-B.

The **prune** procedure checks the generated samples against the given map representation m and removes any samples that travel outside of the road bounds from further evaluation. We also refer to this step as the "lane-checking" procedure.

The **getCosts** procedure calculates the cost for the remaining sample trajectories. We want "cost" to represent "smoothness" of the trajectory from the start state to the end state. That is, we want trajectories that remain close to the center of the lane and far away from the lane boundaries on both sides to have lower cost, and any trajectories that drift close to the lane boundaries or cross lanes many times to have higher cost. This cost function $c(t)$ for a trajectory t is calculated as:

$$
c(t) = Cr_l(t) - MeanDist_l(t)) \tag{9}
$$

where $Cr_l(t)$ represents the number of times the trajectory t crosses a lane boundary on the map, and $MeanDist_l(t)$ represents the average distance of the trajectory from its closest lane boundary on the map. We refer to this as the *smoothness cost*.

Another cost function considered was simply the distance from a *straight-line trajectory* to the goal:

$$
c(t) = \|t - t_{sl}\| \tag{10}
$$

where $\|*\|$ is the L2 norm, and t_{sl} is a straight-line path from the position of initial state x_0 and the position of the goal state x_g. We refer to this as the *straight-line cost*.

Finally, the trajectory with the minimum cost is selected. The steps for this lattice-based search procedure are presented in Algorithm 1.

Algorithm 1: General Lattice-Based search procedure.

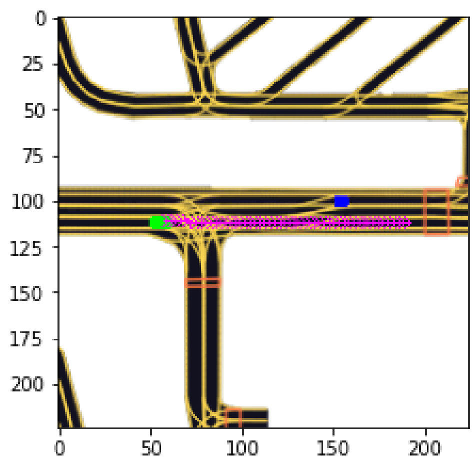
```
Result: optimal trajectory
Inputs:
    $x_0$ start state of vehicle
    $x_g$ goal states of vehicle
    $n$ number of samples
    $m$ map;
sampleTrajectories = getSamples($x_0$, $x_g$, $n$);
sampleTrajectories = prune(sampleTrajectories, $m$);
trajectoryCosts = getCosts(sampleTrajectories, $m$,
  $x_0$, $x_g$);
bestTrajectory = argmin(trajectoryCosts);
return bestTrajectory;
```

A key assumption of this implementation is that the ego vehicle and surrounding vehicles are moving at a constant, uniform velocity. This makes obstacle avoidance easier by avoiding relative velocity differences, allowing our ego vehicle to focus on following the road and staying in the lane boundaries. However, the downside to this assumption is that the algorithm would not be safe if deployed in the real world. In order to move past this assumption and make the algorithm more robust, we would need to predict the path and behavior of neighboring vehicles and incorporate this information into ego vehicle planning, making our planner spatiotemporal. However, we feel that this assumption is valid for our case, since the lattice planner is just one part of the planning stack. In a real world implementation, predictions for the surrounding vehicles would be computed higher in the stack and passed to the lattice planner as an input. Additionally, if surrounding vehicles deviated from this path, a reactive planner would handle the aberration.

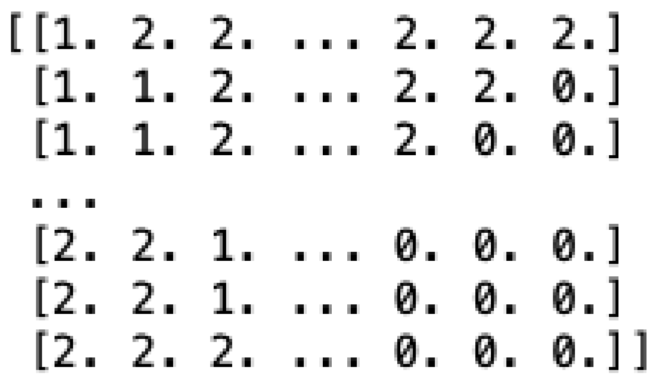
IV. Experiments

A. Deep Learning Pipeline

We implement the Lyft baseline solution [6] that uses a ResNet50 CNN architecture to plan trajectories for the ego



(a) Map image from Lyft Prediction dataset



```
[[1. 2. 2. ... 2. 2. 2.]
 [1. 1. 2. ... 2. 2. 0.]
 [1. 1. 2. ... 2. 0. 0.]
 ...
 [2. 2. 1. ... 0. 0. 0.]
 [2. 2. 1. ... 0. 0. 0.]
 [2. 2. 2. ... 0. 0. 0.]]
```

(b) Extracted matrix representation of the map

Fig. 2: An example map provided by the Lyft Prediction dataset and corresponding representation used as input to our algorithm. The map on the left depicts the start position of the vehicle as a green rectangle, and purple lines as the target trajectory. Blue rectangles represent other vehicles. In the matrix representation on the right, 1 represents a lane, 2 represents lane and crosswalk boundaries that the vehicle may cross, and 0 represents non-drivable areas.

vehicle by learning directly from the semantic maps. The trajectory is planned over a 1.2 second window. As the data is collected at a frame rate of 10 frames per second, this means we plan a trajectory by predicting 12 future coordinates along which the ego vehicle will traverse, each with an interval of 0.1 seconds between them. The dataset is made up of short driving scenes of about 25 seconds. We build a semantic map for each frame in the scene, encoded with lane boundaries, obstacle masks and traffic light information. A pre-trained ResNet50 CNN model learns to predict a set of future coordinates that the ego vehicle will traverse from the semantic maps. This set of future coordinates form the planned trajectory of the ego vehicle.

B. *Lattice Planning Pipeline*

1) Inputs: We use the semantic maps and map API provided by the Lyft Prediction dataset and L5kit library [3]. Given a map from the dataset, we construct a 2d matrix of values representing road areas, lane boundaries, and non-drivable areas. Figure 2 depicts this extraction. In the matrix representation, a ‘1’ represents a lane, a ‘2’ represents lane and crosswalk boundaries that the vehicle may cross, and ‘0’ represents non-drivable areas. Each map from the L5kit library also provides start and goal positions, and headings for the vehicle from which we can construct a start and goal state. The extracted 2d matrix representation of the map, and the start and goal states are provided as inputs to the Lattice-Based search procedure.

2) Lattice-Based Search Implementation: After implementing a pipeline between the L5kit library and the lattice planner, we adapted code from Coursera Course on Motion Planning for Self-Driving Cars [14] to solve the optimization (7).

Beyond this, we developed our own code for sampling points along the curve given the solved spiral coefficients, accomplished using the cumulative trapezoid function. We also developed our own code to prune trajectories. Pruning was accomplished with a simple “lane-checking” procedure: if any point on the trajectory entered a cell labeled ‘0’, corresponding to outside of road boundaries, then that trajectory was removed. Finally, we implemented the cost functions (9) and (10) to select the best trajectory. An example is shown in Fig. 3.

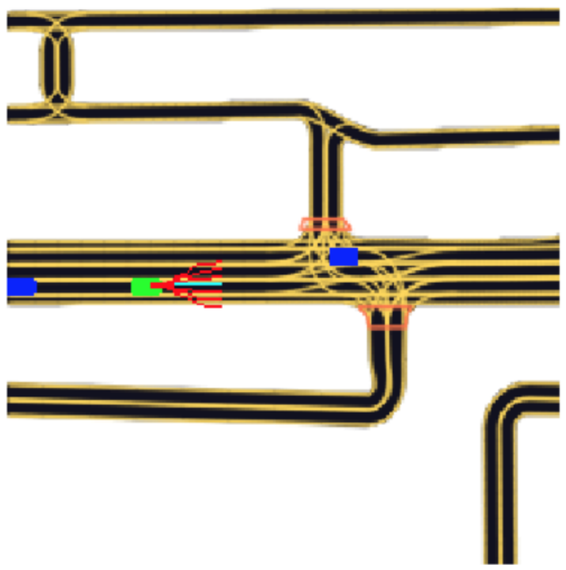


Fig. 3: Lattice Planner example output. Red lines indicate all sampled trajectories. The teal line represents the chosen trajectory (the trajectory with minimum straight-line cost).

C. *Performance Evaluation*

We have set up our experiments as an openloop planning problem, where we plan a trajectory for each frame in the dataset and compare how closely our trajectory matches the ground truth trajectory as compared to the learning-based trajectory. This means we don't actually simulate the ego vehicle to be in its planned position in the next frame, but instead consider the next frame as a different unique initial state. This allows us to make a quantitative evaluation against the ground truth by simply calculating the distance and heading error at each sampled coordinate along the trajectory.

We use 4 metrics to compare our results and the learning-based results against the ground truth:

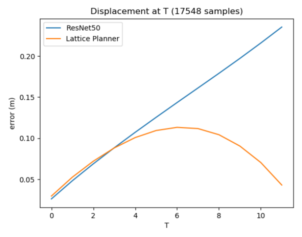
1) Average displacement at T, which shows the average euclidean distance error across all frames between the planned trajectory and ground truth at each of the 12 sampled coordinate points on the trajectory.
2) Average Displacement Error (ADE), which measures the average euclidean distance error along the entire trajectory for each sample.
3) Average angle error at T, which shows the angular distance error between the headings in the planned trajectory and ground truth at each of the 12 sampled coordinate points on the trajectory averaged over all frames.
4) Average Angle Error (AAE), which measures the average angular distance error along the entire trajectory for each sample.

V. Results and Discussion

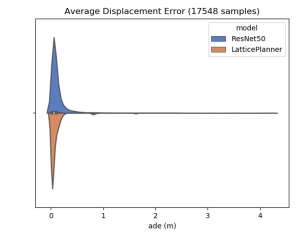
We compared the Deep Learning and Lattice Planning methods both with and without the "lane-checking" or **prune** step, and with both cost functions (9) and (10), using sample maps from the Lyft dataset. We found that with the lane-checking step, the Lattice Planner would yield larger displacement errors than without the lane-checking step. This was the case when using both the *straight-line* and *smoothness* cost function. We believe the discretization of the map when extracting the matrix representation resulted in clipping of lane areas, which led to some valid trajectories to be pruned out because they seemed to go out of lane bounds.

Additionally, we found that lower cost did not always equate to lower path displacement or path angle error. We believe this is because our cost functions, while defined according to what we believe would represent the "best" trajectory, did not represent the true human-driven trajectories from the dataset. This is evidenced by comparing the average cost of the Deep Learning and Lattice Planner generated trajectories with the average ground truth cost of the human trajectories. For example, using the smoothness cost (9) as the cost function with lane-checking would yield lower cost for the Lattice Planner than the Deep Learning Planner and the human trajectory, but the Lattice Planner would yield much higher displacement error. Using the straight-line cost (10) yielded smaller displacement and angle errors than using the smoothness cost (9) for the Lattice Planner when lane-checking is turned on. Overall average path displacement error, average path angle error, average path cost, and average path computation time are presented in Table I. The Lattice Planner had faster computation time than the Deep Learning Planner.

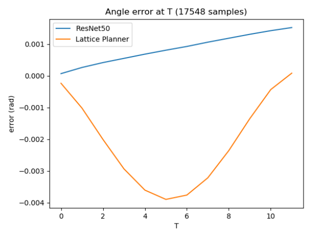
Upon further analysis using the straight-line cost function, we found that average displacement error over time for the Lattice Planner would initially be higher than that of the Deep Learning Planner, but then gradually decrease as errors for the Deep Learning Planner continue to linearly increase. With lane checking off, average angle error over time would similarly be higher for the Lattice Planner initially before gradually decreasing. We found that adding lane checking reduced the angle error over time for the Lattice Planner at the expense of small increase in displacement error over time. Figures 4 and 5 show the average displacement error and average angle error over time, as well as ADE and AAE distributions for both the Deep Learning Planner and the Lattice Planner. The results presented in Figure 4 were generated using the full Lyft dataset, while the results reported in Figure 5 use the same 501 samples as in I.



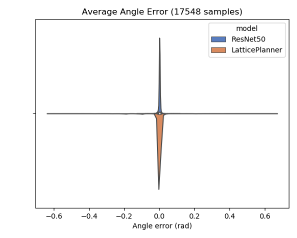
(a) Displacement Error over time



(b) Average Displacement Error



(c) Angle Error over time



(d) Average Angle Error

Fig. 4: Angle and Displacement Errors for Deep Learning (ResNet50) Planner and Lattice Planner using straight-line cost function with lane-checking on.

VI. Conclusion

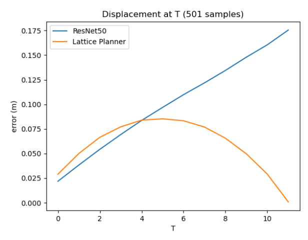
We implemented a Lattice Planner to generate short-horizon paths for an autonomous vehicle using data from the Lyft Prediction dataset. We evaluated the Lattice Planner against a pre-trained Deep Learning Planner by comparing generated trajectories against ground truth data collected from human drivers. Using a straight-line cost function, the paths generated by the Lattice Planner showed lower displacement error as compared to the Deep Learning Planner, and also had faster execution time.

Future work includes extending the Lattice Planner horizon for longer trajectory planning, and incorporating dy-

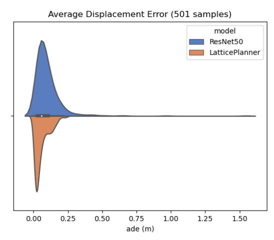
		Smoothness Cost Function				
		Path Displacement Error	Path Angle Error	Compute Time	Cost	Ground Truth Cost
Deep Learning	-	0.101	0.002	0.147	-1.752	-1.733
Lattice Planner	Lane Checking	0.496	-0.053	0.048	-2.432	
	No Lane Checking	0.058	0.003	0.046	-1.729	

		Straight Line Cost Function				
		Path Displacement Error	Path Angle Error	Compute Time	Cost	Ground Truth Cost
Deep Learning	-	0.101	0.002	0.151	4.204	4.141
Lattice Planner	Lane Checking	0.079	0.000	0.047	4.222	
	No Lane Checking	0.058	0.003	0.045	4.106	

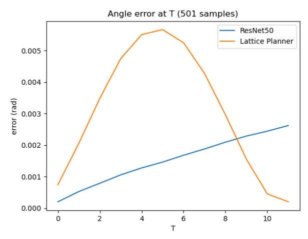
TABLE I: Comparison of the Deep Learning Planner and Lattice Planner (with lane-checking on and off) for the smoothness cost function (9) and the straight-line cost function (10) over 501 samples.



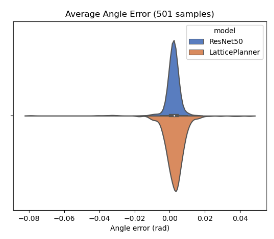
(a) Displacement Error over time



(b) Average Displacement Error



(c) Angle Error over time



(d) Average Angle Error

Fig. 5: Angle and Displacement Errors for Deep Learning (ResNet50) Planner and Lattice Planner using straight-line cost function with lane-checking off.

namic collision checking in the planning procedure using a spatiotemporal planner.

Acknowledgments

We thank the Lyft Level 5 team for open sourcing their dataset.

References

[1] N. Kalra and S. M. Paddock, "Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?" *Transportation Research Part A: Policy and Practice*, vol. 94, pp. 182–193, 2016.

[2] M. Schwall, T. Daniel, T. Victor, F. Favaro, and H. Hohnhold, "Waymo public road safety performance data," *arXiv preprint arXiv:2011.00038*, 2020.

[3] J. Houston, G. Zuidhof, L. Bergamini, Y. Ye, A. Jain, S. Omari, V. Iglovikov, and P. Ondruska, "One thousand and one hours: Self-driving motion prediction dataset," *arXiv preprint arXiv:2006.14480*, 2020.

[4] J. Ziegler and C. Stiller, "Spatiotemporal state lattices for fast trajectory planning in dynamic on-road driving scenarios," *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*.

[5] D. Madas, M. Nosratinia, M. Keshavarz, P. Sundstrom, and R. Philippsen, "On path planning methods for automotive collision avoidance," *2013 IEEE Intelligent Vehicles Symposium (IV)*, pp. 931–937.

[6] Lyft, "l5kit," https://github.com/lyft/l5kit/tree/master/examples/planning, 2020.

[7] N. Djuric, V. Radosavljevic, H. Cui, T. Nguyen, F.-C. Chou, T.-H. Lin, N. Singh, and J. Schneider, "Uncertainty-aware short-term motion prediction of traffic actors for autonomous driving," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2020, pp. 2095–2104.

[8] W. Zeng, S. Wang, R. Liao, Y. Chen, B. Yang, and R. Urtasun, "Dsdnet: Deep structured self-driving network," in *European Conference on Computer Vision*. Springer, 2020, pp. 156–172.

[9] S. Mozaffari, O. Y. Al-Jarrah, M. Dianati, P. Jennings, and A. Mouzakitis, "Deep learning-based vehicle behavior prediction for autonomous driving applications: A review," *IEEE Transactions on Intelligent Transportation Systems*, 2020.

[10] M. Pivtoraiko, R. Knepper, and A. Kelly, "Differentially constrained mobile robot motion planning in state lattices," *Journal of Field Robotics*, vol. 26, no. 1, pp. 308–333, 2009.

[11] M. McNaughton, C. Urmson, J. Dolan, and J. Lee, "Motion planning for autonomous driving with a conformal spatiotemporal lattice," *2011 IEEE International Conference on Robotics and Automation*.

[12] D. González, J. Pérez, V. Milanés, and F. Nashashibi, "A review of motion planning techniques for automated vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 4, pp. 1135–1145, 2015.

[13] C. Katrakazas, M. Quddus, W. Chen, and L. Deka, "Real-time motion planning methods for autonomous on-road driving: State-of-the-art and future research directions," *Transportation Research Part C: Emerging Technologies*, vol. 60, pp. 416–442, 2015.

[14] S. Waslander and J. Kelly, "Motion planning for self driving cars," *Coursera Course, University of Toronto*.

[15] A. Piazzi and C. Guarino Lo Bianco, "Quintic g2 splines for trajectory planning of autonomous vehicles," *2000 IEEE Intelligent Vehicles Symposium*.

[16] A. Kelly and B. Nagy, "Reactive nonholonomic trajectory generation via parametric optimal control," *The International Journal of Robotics Research*, vol. 22, no. 7, pp. 583–601, 2015.