Reinforcement Learning Approaches for Racing and Object Avoidance on AWS DeepRacer

Jacob McCalip
Texas State University
jsm246@txstate.edu

Mandil Pradhan
Texas State University
jns176@txstate.edu

Kecheng Yang
Texas State University
yangk@txstate.edu

***Abstract*—Developing autonomous driving models through reinforcement learning is gaining widespread prominence. However, a pervasive problem is developing obstacle avoidance systems. Specifically, optimizing path completion times while avoiding objects is an underdeveloped area of research. AWS DeepRacer's platform provides a powerful architecture for engineering and analyzing autonomous models. Using AWS DeepRacer, we integrate two pathfinding algorithms, A* and Line-of-Sight (LoS), into this paradigm of autonomous driving. LoS is a novel algorithm that incrementally updates the model's heading angles to amply reach its destination. We trained three types of models: Centerline, A*, and LoS. The Centerline model utilizes logic from AWS and is practically the only model used by the AWS DeepRacer community that avoids objects. We developed models from A* and LoS that outperformed the default models in time per lap while maintaining commensurate stability.**

I. Introduction

Autonomous robots have revolutionized numerous sectors of our economy and are utilized in a wide range of industries and applications, including manufacturing, warehouses, and healthcare. These machines are capable of performing tasks without direct human supervision, relying on a combination of sensors, learning-enabled algorithms, computer vision techniques for perceptions, and real-time data to make decisions and execute tasks. Among these autonomous machines, autonomous vehicles have accentuated focus for both research and development. Self-driving cars are a prime example of autonomous vehicles that navigate through traffic, utilizing sensors, cameras, and other technologies to detect objects and obstacles, determine optimal routes, and make safe and efficient decisions on the road.

While still facing many challenges to overcome towards full-scale adoption, autonomous driving is poised to bring substantial benefits to our society. One of the most significant benefits of deploying self-driving cars is the potential to largely reduce traffic accidents. Over 90 percent of road accidents are caused by some degree of human error, including distraction, impaired driving, and poor decision-making. It is believed that, with the advances in autonomous driving technologies, the number of accidents should plummet [9]. Therefore, reliable, effective, and efficient learning-enabled algorithms and software implementation are essential to autonomous driving, which can revolutionize the transportation industry. In this paper, our objective is to optimize collision avoidance through the lens of reinforcement learning.

This work is supported in part by NSF grants CNS-2104181 and CNS-2149950, and a REP grant from Texas State University.

Related work. Learning-based autonomous driving can be traced back to 1991, when artificial neural networks are applied in the ALVINN system [7]. More recently, A*-like pathfinding algorithms in continuous environments for autonomous vehicles were investigated [3]. To enable comparative research, benchmark suites for autonomous driving, such as KITTI [4], were developed. In the AWS League, the main goal among the thousands of competitors is to design a model which circles a lap around a given track in the shortest amount of time possible. A fairly common approach among the AWS DeepRacer Community is to use the K1999 Path-Optimization Algorithm. This algorithm is not commonly used in machine learning or autonomous driving and has little documentation. However, "It works by iteratively decreasing the line's curvature" [5]; this helps racers to cut down the overall time per lap. A less commonly used approach is to use a path-finding program that increases the radius around curves. Zhu et al. created a pathfinding algorithm to implement this concept, however, it is not clear how to reproduce their work [11].

Contributions. In this research, we prototype and evaluate collision avoidance approaches on the AWS DeepRacer platform. The AWS DeepRacer platform provides a simulation environment for training models, which can be deployed on physical model cars. We focus on investigating reinforcement learning models, which are trained by exploring the environment and striving for optimal results with respect to the rewards that are defined differently in different models. In particular, we

- design and implement A* and Line-of-Sight (LoS) approaches to train models that attempt to provide optimal paths;
- enhance their navigational capabilities by integrating object avoidance methods; and
- demonstrate that the models trained by these approaches outperform those by the AWS DeepRacer default approach in terms of learning efficiency and racing quality.

II. Background

This research focuses on reinforcement learning, which is based on the concept of *reward*. Intuitively, a model learns through a trial-and-error process to maximize its reward. Over time, it is expected to yield a model that is adapted to its environment, as the learning process favors the actions and behaviors that lead to the highest reward in a reinforced manner. A key concept in reinforcement learning is the distinction

between exploratory and exploitative behaviors. Exploration involves taking random actions regardless of whether the results would be favorable or not. On the other hand, exploitation involves using information already gathered in prior attempts to move towards desirable results.

AWS DeepRacer. AWS DeepRacer is an ecosystem for investigation and research on autonomous driving technologies. Developed by AWS, DeepRacer is centered around a 1/18th scale racing car as shown in Fig. 1(a). The racing of AWS DeepRacer can be done either in the simulation or on a physical track. Fig. 1(b) shows an example physical track we built in our research lab. Furthermore, having been open-sourced, DeepRacer has fostered a number of projects beyond its default framework.

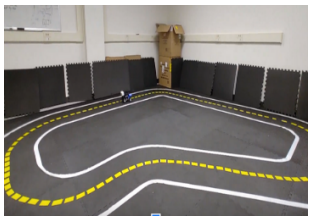
Our efforts mainly focus on the DeepRacer simulation environment. Further, we used the simulation to concentrate more on the computation and control approaches, while limiting unpredictable impacts of the physical world. This isolation of variables is imperative for conducting systematic experiments and data analysis. The official simulation environment is via the AWS DeepRacer console provided by AWS. However, in recent years, a community simulation environment has been developed and widely adopted, which is called DeepRacer for Cloud (DRFC). This tool is a loosely bound set of scripts that allows for model training compatible with AWS's official server and league. An advantage of DRFC is that we have more freedom to adjust the simulation to suit our desires. DRFC uses Docker to house Minio, Robomaker Sagemaker, RL Coach, and Gazebo, of which the simulation environment is composed. The function of each of these components is briefly explained as follows.

- *Minio* is an object storage system.
- *AWS Robomaker* provides service to develop and test robotic applications.
- *AWS Sagemaker* provides infrastructure for building learning models.
- *RL Coach* is developed by Intel for fine-tuning algorithm development.
- *Gazebo* provides simulation of 3D physics.

Simulation environment. Hyperparameters are critical for training models. In the simulation, hyperparameters are defined from the outset, instead of being learned from data, and help to shape the model's behaviors. In the context of DeepRacer, we have the following major hyperparameters [2].

- `batch_size`: gradient descent batch size, which determines sample size that is taken into consideration for updating the training model.
- `num_epochs`: number of times the training data set will be processed in loop to update the learning parameters.
- `discount_factor`: determines the importance of immediate vs future rewards.
- `beta_entropy`: randomness of policy distribution.
- `lr`: the step function of gradient descent.
- `loss_type`: provides the difference between actual and predicted results of a model.

(a) Physical car



(b) Physical track

Fig. 1. Physical environment.

- `num_episodes_between_training`: defines how frequently to update the policy of the agent.

Using local simulation via DRFC, there is a greater degree of control. To illustrate, in local training, we may switch the direction of the car (clockwise vs counterclockwise), change the lighting of the simulation, change the location, type, and amount of objects on the track, etc. Moreover, Amazon is restrictive of Python packages, such as the time library. Consequently, local simulation is a standard practice among the AWS DeepRacer community. Thus, we utilized a local simulation environment for this study.

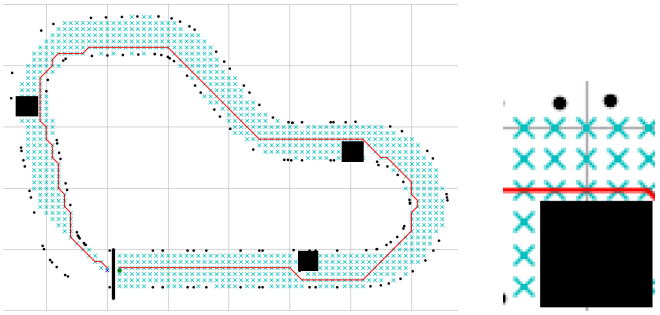
Models and rewards functions. A model is mainly comprised of action space, reward function, and metadata information. The action space is the set of all possible actions the agent may take. In our context, we have the option of choosing a discrete or continuous action space. A discrete action space means that the possible actions are finite and countable. Conversely, a continuous action space can take on an infinity of values within a certain range. It is well known that continuous action spaces tend to take longer to converge, however, they are also less prone to human bias.

The reward function is at the core of reinforcement learning. The reward function maps the behavior of the model's state and action to an associated reward. which indicates the agent's performance. DeepRacer utilizes Python to write a reward function and possess simulation parameters used to feed the model information about its environment. This information, in a vague sense, can be used to incentive the model to produce desired outcomes. For instance, a parameter of the simulation called "distance_from_center" returns the distance in meters from the center of the track. The reward function serves as the fundamental logic guiding the nature of the model.

Physical build: Sim2Real. Models trained from simulation can be uploaded to the physical car and used on an actual track such as the one we built in Fig. 1. DeepRacer is fairly unique in that we can take models trained in simulation to a physical counterpart; other popular autonomous driving simulations such as CARLA have no physical counterparts. Moreover, work done by Revell et al. describes procedures to optimize the bridging of the Sim2Real gap on AWS DeepRacer [8].

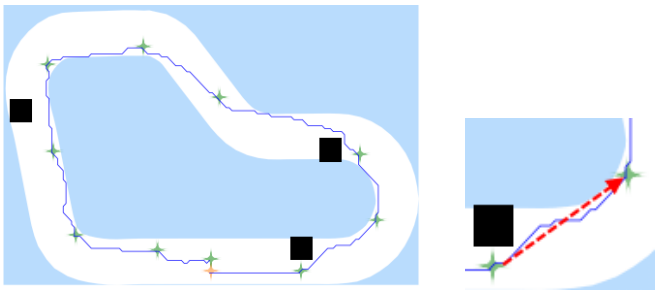
III. Investigated Approaches

In addition to DeepRacer's default approach (Centerline), we develop, implement, and investigate two new approaches (A-Star and Line-of-Sight) to obtain models that are capable



(a) Holistic path generation (b) Zoom-in

Fig. 2. A-Star path generation.



(a) Holistic path generation (b) Zoom-in

Fig. 3. Line-of-Sight path generation.

of racing when obstacles are present. We briefly explain the three approaches as follows.

Centerline. The Centerline algorithm created by Amazon utilizes the centerline of the track and the location of objects to guide the model; the further the model is from the centerline, the less reward; on the other hand, if the model is in the same lane as the object and the model gets too close to the object, the car gets progressively less reward [1].

A-Star. One of the approaches used is the A* path-finding algorithm, which generates a predefined path that avoids obstacles in the track. It serves as a guide to incentivize the model to follow the path. In Fig. 2, the cyan 'X' points are the searched nodes in the grid; the outermost two dots represent the track border, and the black box represents an object. Moreover, the red line is Fig. 2 is the final path calculated from A*. It should be noted that in Fig. 2 there is a small gap in the line, however, this gap is only for present in visualization, not in the actual path file generated. The object locations used in Fig. 2 is representative of our training. Related, we have Fig. 4, a heatmap, the brighter, more dense clusters of orange colors here highlight a higher reward for a given area. This approach is considered static because the generated path remains unchanged throughout the training period. By incorporating this approach into our training process, the model converges faster, ultimately resulting in reduced training time.

Line-of-Sight. The Line-of-Sight (LoS) algorithm utilizes the vehicle's field of vision to identify the furthest destination on the race track and then generates a path and heading angle toward that location. In our experiment, we set a limit on how far the vehicle could see. As the vehicle progresses, it receives updated paths and heading angles to guide its movements. This approach is considered dynamic because it provides regular updates to the model during the training period. However, one limitation of this approach is that it may take longer for the model to converge since the path and heading angle depend on the vehicle's location and field of vision, which may vary. The current location of the vehicle plays a significant role in determining the heading angle and destination point on the track.

IV. Experiments and Evaluation

Experiment settings. To standardize our finding, we trained each model for 0.5 hours, 1.5 hours, and 3 hours; with each using 4 workers; training with multiple workers enables parallelization of the training process, thus, a trained model will converge faster in a given period.

All models were trained with the stereo camera and a continuous action space on the "re:invent 2018" track:

- speed: $1.0 \sim 1.5$
- steering angle: $-20 \sim 20$

The hyperparameters in our experiments are summarized as follows:

batch_size = 64	stack_size = 1
num_epochs = 3	epsilon_steps = 10000
discount_factor = 0.985	e_greedy_ value = 0.0003
beta_entropy = 0.01	term_cond_max_episodes = 1000
lr = 0.0003	term_cond_avg_score = 350
loss_type = huber	exploration_type = categorical
num_episodes_between_training = 32	

These settings are slightly modified from the default in order to synergize better with continuous models [10].

- The Centerline models utilize reward 4 in the DeepRacer developer's guide which involves avoiding stationary objects [1].
- The A-Star models utilize a custom reward we made designed to follow A*'s path, with python helper functions borrowed from Daniel Gonzalez [5].
- The Line-of-Sight models utilize a line-of-sight algorithm to dynamically determine the vehicle's optimal path, destination, and heading angle to follow.

All models were trained with three stationary cars, which act as obstacles for the model to avoid.

Once the training was complete, we ran an evaluation for 50 iterations on each model, this data is used for our analysis. It should be noted that the evaluation was performed on the best checkpoint of each model, which here is defined as the checkpoint with the highest reward. Fig. 4 illustrates this process by showing an example heat map of event rewards during the training of A-Star models.

Results and evaluation. We first report the training time for each model to *converge*, which is defined as the training time for a model to reach its best mean lap completion time as follows:

- Centerline converges at 3 hours of training with a mean lap completion time of 14.27 seconds;

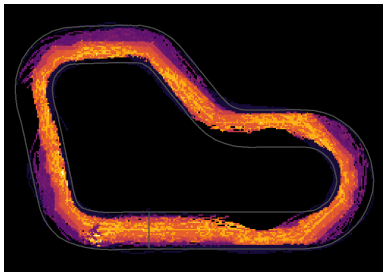


Fig. 4. Heatmap of the accumulation of event rewards during training.

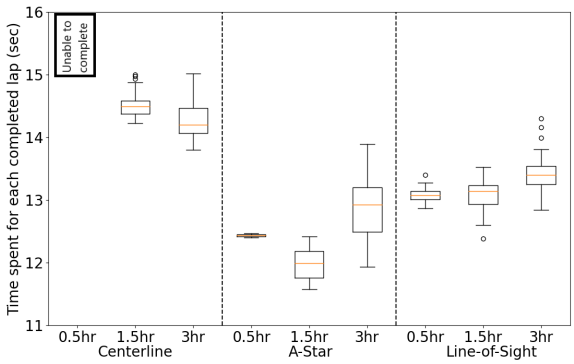


Fig. 5. Time spent on completed laps to show the racing speed of models.

- A-Star converges at 1.5 hours of training with a mean lap completion time of 11.98 seconds;
- Line-of-Sight converges at 0.5 hours of training with a mean lap completion time of 13.08 seconds.

These results demonstrate that our proposed approaches need less training time to reach a model with optimal performance than the default Centerline approach.

In Fig. 5 we show each type of model trained on 0.5 hours, 1.5 hours, and 3 hours of training (x-axis). We did not increment one training to the next, instead, each model was trained independently. With 0.5 hours of training, Centerline could not complete the evaluation, A-Star had only 3 completed laps within the evaluation, but Line-of-Sight had a complete 50 out of 50 completed laps at this time; this small amount of training also proved to be Line-of-Sight's fastest time compared to longer training sessions as shown in Fig. 5. The nature of Line-of-Sight promotes rapid adaptation to the model's local environment. In Fig. 5, we show that models from A-Star and Line-of-Sight outperformed the standard models in speed in every instance. We have included a video playlist showing all the evaluations conducted in this study for demonstration [6].

V. Conclusion

In this work, we leverage the AWS DeepRacer platform to investigate reinforcement-learning-based approaches for autonomous driving. The goal is to create models that are able to reach faster racing speeds while maintaining reasonably reliable performance. In addition, objects may exist in the track and should be avoided by the racing car. By integrating the concepts of A* and LoS algorithms, we created novel

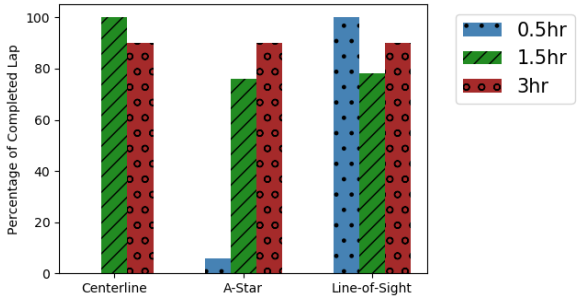


Fig. 6. Percentage of laps that are perfectly completed by models.

solutions inside DeepRacer's environment that we believe has the potential to transcend to other domains of reinforcement learning. We developed models from these two algorithms that have similar stability to the default models but achieved faster performance.

Future work. In subsequent work, we aim to delve deeper into the hyperparameter settings, and how that impacts the training behavior of the model. This future research should display how each setting affects the model's performance. Furthermore, we plan to investigate the application of the trained model in the physical environment. Namely, we have observed that there is performance variability between the simulation and the physical environment. This discrepancy warrants further investigation to identify the factors that contribute to bridging the Sim2Real gap.

References

[1] Amazon. Aws deepracer developer guide. Online at https://docs.aws.amazon.com/deepracer/latest/developerguide/deepracer-reward-function-examples.html.

[2] Siddhartha Banerjee. Aws deepracer — looking under the hood for design of the reward function and adjusting hyperparameters. Online at https://medium.com/analytics-vidhya/aws-deepracer-looking-under-the-hood-for-design-of-the-reward-function-and-adjusting-e9dd3805ebbf.

[3] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo, and James Diebel. Path planning for autonomous vehicles in unknown semi-structured environments. volume 29, page 485–501, USA, apr 2010. Sage Publications, Inc.

[4] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3354–3361, 2012.

[5] Daniel Gonzalez. An advanced guide to aws deepracer. Online at https://towardsdatascience.com/an-advanced-guide-to-aws-deepracer-2b462c37eea.

[6] Jacob Mccalip. Playlist of model evalauations. Online at https://www.youtube.com/playlist?list=PLh53BF3bZA6rR2tk3GCcLMVeN8Cy-95Rc.

[7] Dean A. Pomerleau. Efficient training of artificial neural networks for autonomous navigation, 1991.

[8] Jacob Revell, Dominic Welch, and James Hereford. Sim2real: Issues in transferring autonomous driving model from simulation to real world. In *SoutheastCon 2022*, pages 296–301. IEEE, 2022.

[9] Santokh Singh. Critical reasons for crashes investigated in the national motor vehicle crash causation survey, Feb 2015.

[10] Boltron Racing Team. Continuous action space, reward func, and hyperparameters for top 15 finish in deepracer! Online at https://youtu.be/11Sta3idwZI.

[11] Wenjie Zhu, Haikuo Du, Moyan Zhu, Yanbo Liu, Chaoting Lin, Shaobo Wang, Weiqi Sun, and Huaming Yan. Application of reinforcement learning in the autonomous driving platform of the deepracer. In *2022 41st Chinese Control Conference (CCC)*, pages 5345–5352. IEEE, 2022.