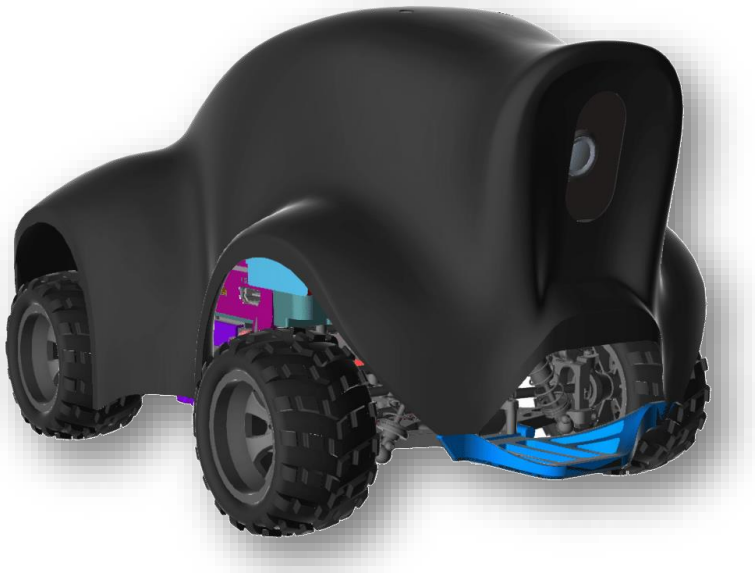
Design reward function for RL car agent



Ali Ghasemi

Mechanical Engineering Student

Research Enthusiast in Robotics and AI

Iran University of Science and Technology

Summer 2024

Contents

Abstract

This project involves the development of a custom reward function to enhance the performance of an autonomous vehicle in the AWS DeepRacer simulation environment. The reward function aims to optimize the vehicle's adherence to the center line, speed maintenance, and overall track positioning to achieve faster lap times and improved stability. This document details the methodology, implementation, results, and key insights gained from the project.

Key Skills: Reinforcement learning, Python

Introduction

Project Overview

AWS DeepRacer is an educational platform by Amazon Web Services (AWS) that provides an accessible introduction to reinforcement learning (RL) through autonomous racing. Participants train a model to navigate a virtual track, with the goal of achieving the fastest lap time while keeping the car on track. The model's behavior is guided by a reward function, which is a key component in reinforcement learning that incentivizes desirable actions and penalizes undesirable ones.

In this project, we focus on designing and implementing a custom reward function aimed at optimizing the car's performance in the AWS DeepRacer environment. The reward function is designed to balance multiple factors, such as staying close to the center line of the track, maintaining optimal speed, and reducing the likelihood of the car veering off the track. By tuning these factors, the goal is to enhance the overall stability and speed of the car, leading to better performance in the simulated racing environment.

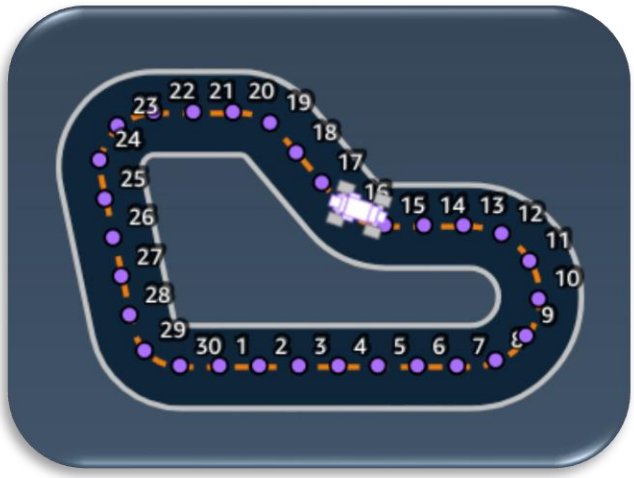
Motivation:

As a mechanical engineering student fascinated by robotics and mechatronics, I have a strong interest in the intersection of mechanical systems and intelligent control. My academic background has equipped me with a solid understanding of these fields, and I've spent time studying tutorials on reinforcement learning (RL) to build my knowledge in this area, particularly in autonomous systems like self-driving cars.

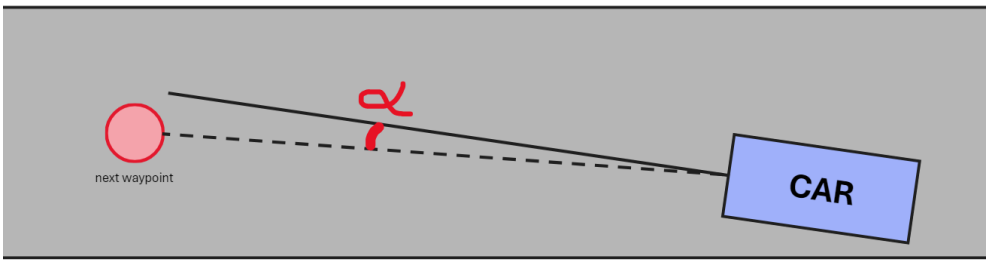
This project provides an opportunity to apply what I've learned about RL to a real-world challenge. By developing a reward function for the AWS DeepRacer, I can bridge the gap between my mechanical knowledge and the software-driven aspects of AI. My interest in self-driving cars makes this project particularly exciting, offering insights into the technologies shaping the future of transportation. Through this project, I aim to deepen my understanding of how intelligent systems can be trained to make autonomous decisions, an area that is both academically stimulating and relevant to my career aspirations.

Methodology

The path consists of several waypoints, each containing multiple (x, y) coordinates that mark significant milestones along the track.



By knowing the robot's current X, Y position, we can determine the next waypoint as the immediate destination and configure the heading angle. This allows us to calculate the required angle adjustment (alpha) for the robot.



After calculating alpha, this value can be used as a key criterion for training the model. Additionally, the car's speed can be factored in; for example, if the car's speed exceeds a determined threshold, achieving the best lap time is rewarded more efficiently.

Furthermore, we ensure that all wheels remain on the track. If any wheel goes off the road, the model is penalized severely with a substantial negative reward.

Implementation and Train the model

Python Code

Code	Explanation
`import math` `def reward_function(params):`	Due to the necessity of leveraging some mathematical function, math library is imported. Then the name of function and input* is given.

`waypoints = params['waypoints']`
`closest_waypoints = params['closest_waypoints']`
`heading = params['heading']`
`X = params['x']`
`Y = params['y']`
`speed = params['speed']`
`track_width = params["track_width"]`
`distance_from_center =`
`params['distance_from_center']`
`all_wheels_on_track = params['all_wheels_on_track']`
	From all parameters a few of them which are required to implement and satisfy methodology are defined as variables.
	Milestone of track(waypoints)
	Closest waypoint [x and y]
	X and Y locations (m)
	Speed of car (m/s)
	Track width
	Are all wheels on track
	Distance from center of road

`next_point = waypoints[closest_waypoints[1]]` `prev_point = (X, Y)`	Assign current and next desired waypoint in two variables
`reward = 1.0`	Initialize the primary reward

`track_direction = math.atan2(next_point[1] -`
`prev_point[1], next_point[0] - prev_point[0])`
 	
`track_direction = math.degrees(track_direction)`
 	
`direction and the heading direction of the car`
`direction_diff = abs(track_direction - heading)`
 	
`if direction_diff > 180:`
`direction_diff = 360 - direction_diff`
	Calculate the direction to next waypoint.
	Then the radian magnitude is converted to degree.
	Thirdly, code calculates the how much direction of is veered rather than desired direction.
	As we are working in range of [-pi pi] we check and rectify invalid direction deviation

`DIRECTION_THRESHOLD = 10.0`
`SPEED_THRESHOLD = 1.0`
`DISTANCE_PARAMETER = 1 -`
`distance_from_center/track_width/2`
	Define thresholds and criteria needed for calculating reward

`if direction_diff > DIRECTION_THRESHOLD:`
`reward -= 0.4`
`if DISTANCE_PARAMETER <=0.4:`
`reward -= 0.1`
`if speed < SPEED_THRESHOLD:`
`reward -= 0.2`
`if not all_wheels_on_track:`
`reward = -3`
 	
`return float(reward)`
	Penalties are set in this stage. Inn this part if car is deviate from path more than threshold will be punished. Not only that, but also or if its speed was under 1.0. If all four wheels goes off, reward will be set to a significant negative number. Finally, reward will be return to the agent for next decisions.

*: params are defined by framework in a dictionary about environment which provided and given to students.

NOTE: spaces and intends are ignored in the code. If you intend to copy, do not forget to follow python intent rules.

Training Parameters

On the platform, there are several adjustable options available. For instance, you can select between two different agent types: PPO and SAC. You can also set the duration of training.

PPO is a good choice for stability and ease of use, especially in simpler environments or when ample training time is available.

SAC is better suited for more complex tasks or when sample efficiency is important, offering robust performance even with shorter training durations.

The duration of training should be balanced based on the agent's learning curve. While longer training durations generally lead to better performance, it's important to consider potential diminishing returns.

Result

After training my agent using the Proximal Policy Optimization (PPO) method for 10 hours, these are the results achieved in the race:

- **Last update:** 17 August 2024
- **Best lap time:** 1:00.192
- **Average lap time:** 1:00.487
- **Off-track occurrences:** 2 times

Although my best lap time is 18 seconds slower than the world rank 1, my model's performance placed within the top 20% in this race.

Conclusion and Relevant Linkes

Upon completing this project, I gained invaluable experience in designing reward functions for reinforcement learning agents. However, the most significant aspect of this project was applying my knowledge to a real-world scenario, bridging the gap between theoretical understanding and practical implementation.

You can access to python code via my GitHub repository.

You also can connect me through LinkedIn and Gmail.

Gmail: alivghasemi@gmail.com

LinkedIn: www.linkedin.com/in/alivghasemi/