UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Facultad de Ingeniería

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS
Acreditación Institucional de Alta Calidad

Workshop 1

Autonomous Racing

Computer Systems Engineering Program

Professor: Carlos Andres Sierra

Kevin Emmanuel Tovar Lizarazo - 20221020068

Jhojan Stiven Aragon Ramirez - 20221020060

April 09, 2025

Contents

1 System Requirements

1.1 Functional Specifications

- **Sensors**:

  The sensor Lidiar, works using lightnings shutting in many directions from the vehicle and measure the distance to the first obstacle. this in 2D top-down environment this implement easily touring for segments in the track.

  In this case for example the car can throw 14 rays in total(can be more if we gonna):

  - 7 to the left of the vehicle in front
  - 7 to the right of the vehicle in front
  - The front counts too, making it a full range of 14 directions
  - Like in the image.

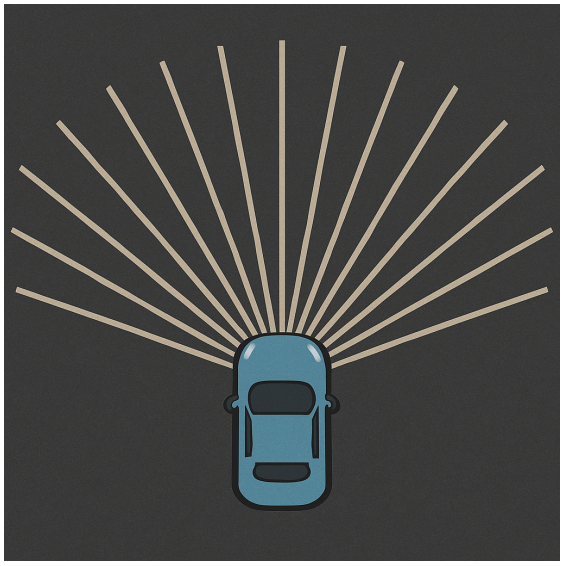

Figure 1: What Lidiar sensor look like

  - Each rays follow a straight line from the vehicle to outside
  - This rays verify if the line crash with some segment to the track.(like edges)
  - The rays calculate the distance from the vehicle to the collision point
  - That distance can save in a slide.

With this in mind, the vehicle ends with a 14 values vector, each indicating which fart is the obstacle more close to that specific direction
Accelerometer Sensor:

* This sensor measures the vehicle's acceleration.
* It works along two or three axes (typically X and Y).
* The sensor gives a vector like $[a_x, a_y]$ for 2D .
* We can use this data to see changes in speed, which helps to know if the vehicle is speeding up, slowing down, or turning.
* The accelerometer reading can help detect collisions or other sudden movements.

– **Actuators:**
In this conduction emulator environment, the actors present effectors of the agents: this are the output of the control model or of the neuronal network. Each actuator corresponds to an action that the agent can apply in the vehicle for modification in his state in the environment. in other word the actors emulate the physical controls in the vehicle:

* steering wheel –¿ direction control
* throttle –¿ front acceleration control
* break –¿ deceleration control

In an emulator 2D, in this case, the actors can model in the next way:

* Steering
  · represent the angle to turn the wheel
  · this range can be from -1.0(turn left) and +1.0(turn right)

* Throttle
  · this represents the amount of force that propels the car forward
  · this range can be 0.0 (does not accelerate) to 1.0(max acceleration)
  · if the environment emulate the max acceleration, this value it's multiplicate for this constants.

* Brake
  · This represents the brake force
  · This range can be 0.0 (without brake) to 1.0(compleat brake)
  · In some environment, can annulate the acceleration or produce negative speed(reverse).

  In the environment GYM these actors are defined through an action_space.

    * Continuo action : In environments like CarRacing, AirSim, Carla o F1TENTH, we can use the continuous action space because the vehicle can have a turn value, acceleration o break in a range.(Zheng et al., 2021)

    * Discreet action : For have more easy the learning, they are some environments of combination controls of action. For example, the action can be:
        · 0 = stay straight
        · 1 = turn left
        · 2 = turn right
        · 3 = accelerated
        · 4 = break

      This makes algorithms like DQD that require a finite number of actions more easy to use.

- **Rewards:**

  The rewards functions are a guide of learning for the agents.In driving, must balance:

  – Advance in the track
    * reward for each checkpoint reached or use distance traveled on the track

    Are example can be has a variable like: reward = progress_along_raceline -1.0

  – penalization for errors
    * When he is out of the track: reward -= 10 or finish season
    * Collision with obstacles: reward -= 5
    * To mush oscillation or unnecessary break
  – Temporal efficient
    * penalization for each frame for encourage the speed trajectory.

    Using the same variable we can make: reward -= 1.0 by frame.

1.2 Use Cases

The agent—a virtual self-driving car—interacts with its environment, which includes the road, track boundaries, curves, intersections and dynamic obstacles. The agent perceives this environment through simulated sensors such as LiDAR and accelerometers . These sensors provide high-dimensional input data that inform the agent about its surroundings.

The primary learning objective is to enable the car to drive autonomously and efficiently across a racing circuit. This includes sub-goals such as staying on the track, avoiding collisions, maintaining optimal speed, and completing laps in minimal time. To achieve this, a reward system is designed to guide learning, often structured to penalize undesirable behavior (e.g., going off-track, collisions), and reward progress and goal completion.

For instance, in OpenAI Gym's CarRacing-v0 environment, the reward function is defined as: "0.1 per frame, and +1000/N for each track tile visited, where N is the total number of tiles. If the car completes the circuit in 732 frames, the final reward is computed as $10000.1 \times 732 = 926.8$ points "(Farama-Foundation, 2022).

Agent-environment interaction follows a continuous loop. At each time step:

1. The agent observes the environment through sensors.
2. Based on these observations, it selects an action (e.g., accelerate, brake, turn).
3. The action affects the environment, resulting in a new state and reward signal.
4. The agent uses this feedback to update its policy in order to improve future decisions.

Through reinforcement learning (RL), the agent gradually improves its policy by maximizing cumulative rewards over time. Adaptation occurs as the agent learns to generalize its behavior across diverse scenarios, such as new track layouts or dynamic obstacles, without explicit reprogramming.

2 High-Level Architecture

2.1 Component Diagram and Data Flow

Component Diagram. This diagram shows the main components of an autonomous driving system that uses reinforcement learning (DQN) inside the CarRacing-v0 simulation environment. The agent receives input from two sensors: a LiDAR sensor that outputs a vector of distances, and an accelerometer that measures the car's acceleration

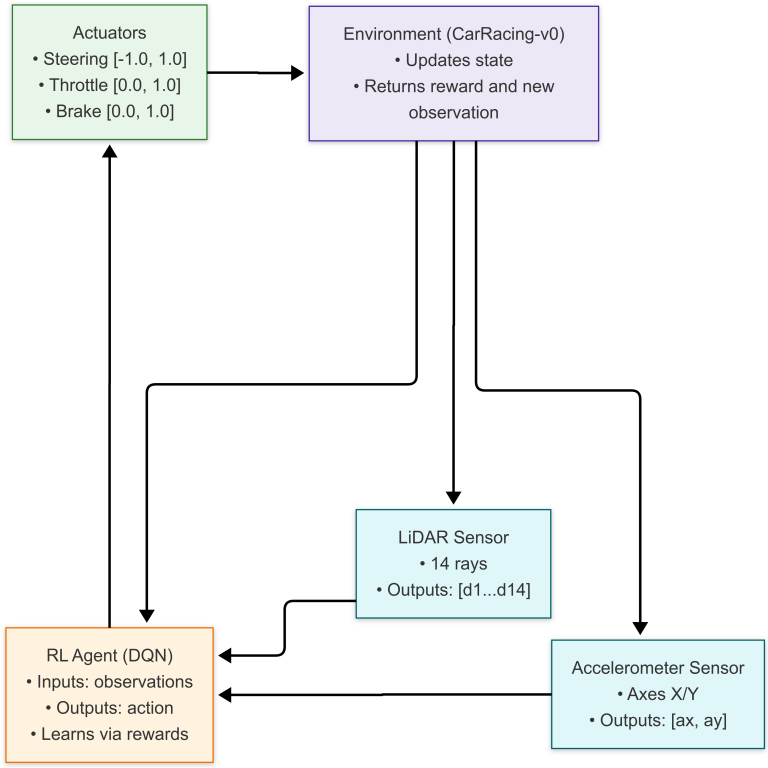

Figure 2: Flow Diagram of the Autonomous Driving System Using Reinforcement Learning

on the X and Y axes. Based on this information, the agent chooses an action (steer, accelerate, or brake), which is sent to the actuators. The environment updates accordingly and returns a new observation and a reward.

2.2 Cybernetic Feedback Loops for Self-Regulation

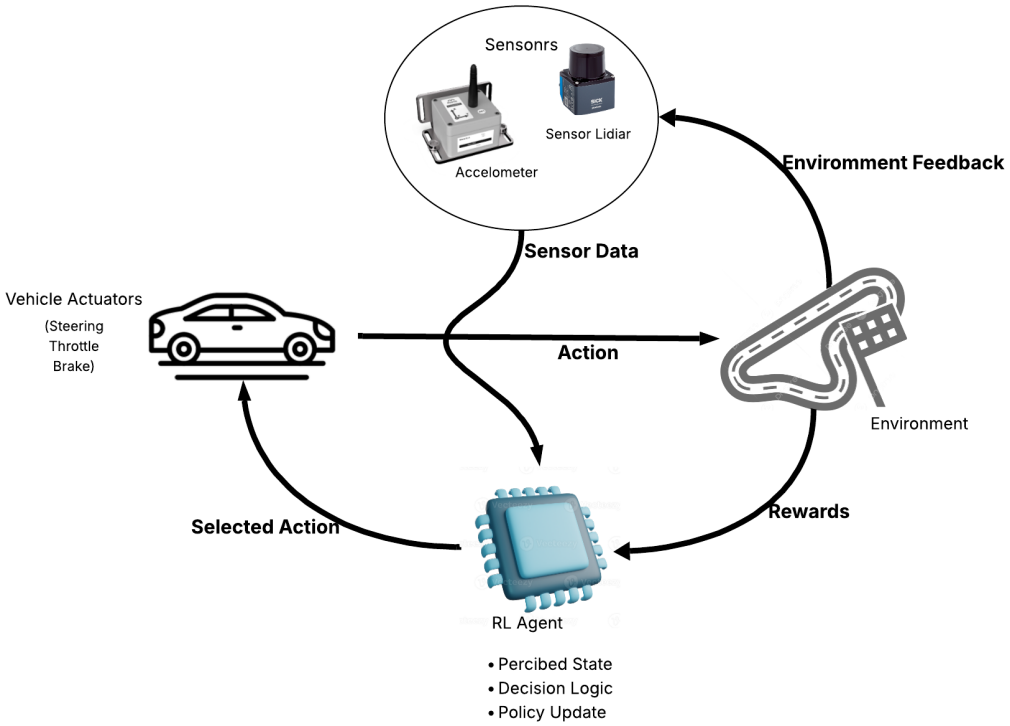

Figure 3: Diagram of Feedback Loops

Feedback Loops. In this cybernetic control loop, the RL agent receives data from the environment through sensors. Based on this input, it constructs a perceived state and, using its decision-making logic (policy), selects an action—such as accelerating, braking, or turning. This action is then executed via the vehicle actuators, altering both the vehicle's state and the surrounding environment. In response, the environment provides new observations along with a reward signal based on the action taken. The RL agent uses this feedback to refine its policy in future iterations, enabling self-regulation and optimal performance over time.

3 Preliminary Implementation Outline

3.1 Selection of Frameworks for Reinforcement Learning

We have two key frameworks well suited for an autonomous driving using the reinforcement learning in simulate environments. This frameworks make more easy the creation of custom environments and implementations of algorithms of RL.

- **Gymnasium**

  Gymnasium provides an interface for reinforcement learning environments. this allows a simple way to define the functions reset(), step(), as well as action and observation spaces.

  - **Modularity and ease of creating custom environments**: We can design racing circuits 2D using the same API, adding virtual sensors like LiDAR and others.
  - **Extensive ecosystem**: There are many examples and repositories demonstrating Gymnasium integrations with various RL libraries and simulation engines (Pygame, Box2D, PyBullet, etc.)((7enTropy7), 2020)(Pitrified, 2019).

- **Stable-Baselines3**

  A reinforcement learning library that implements popular algorithms such as PPO, DQN, SAC, TD3, and A2C in a consistent, well-structured manner.

  - **Direct integration with Gymnasium**: With this we only need to instantiate the Gymnasium environment and connect it with a Stable-Baselines3 model.
  - **Ease of use and rapid experimentation**: This allows train, evaluate, and save models in just a few lines of code, making it ideal for prototyping different network configurations or RL algorithms.

3.2 Timeline for Transitioning from Q-Learning to Advanced DQN

The next is a timeline for moving from basic Q-learning to more advanced DQN approaches.

Stage	Description
Week 1	Basic Q-learning in a Gridworld environment.
Week 2	Q-learning with discrete states (e.g., CarRacing-v0).
Week 3–4	Feature extraction from sensor data (e.g., LiDAR).
Week 5	Introduction to Neural Networks.
Week 6	Deep Q-Networks (DQN): replacing Q-table with neural nets.
Week 7+	DQN extensions: Experience Replay, Target Networks, etc.

Table 1: Timeline for progressing from basic Q-learning to advanced DQN techniques.

Notes

- **Week 1-2:** Understand Q-tables, reward functions, and agent-environment interactions.
- **Week 3-4:** Start using simulated sensor data (e.g., LiDAR vectors) as state representations.
- **Week 5-6:** Replace Q-tables with neural networks to approximate Q-values.
- **Week 7+:** Improve learning stability using advanced DQN techniques like experience replay and target networks.

References

(7enTropy7), U. M. (2020). Racer_ai: Customizable openai gym environment with ppo agent [Developed using stable-baselines3, PPO, and Imitation Learning with DAgger]. https://github.com/7enTropy7/Racer_AI

Farama-Foundation. (2022). Car racing - gymnasium documentation [Official documentation for the CarRacing environment from the Gymnasium project]. https://gymnasium.farama.org/environments/box2d/car_racing/

Pitrified. (2019). Gym-racer: Openai gym environment of a racing car [Last updated 4 years ago]. https://github.com/Pitrified/gym-racer

Zheng, H., O'Kelly, M., & Sinha, A. (2021). F1tenth gym documentation [Technical documentation for the F1TENTH simulation in OpenAI Gym]. https://f1tenth-gym.readthedocs.io/en/latest/