**CS 7642: Reinforcement Learning and Decision Making** **Project #4**

# DeepRacer: Optimization and Adaptation

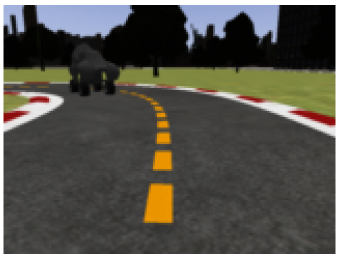
## Contents

## 1 Overview

In this project, you will explore the challenges of training agents for an autonomous racing car using a local variant of AWS DeepRacer (Balaji et al. 2019; Balaji et al. 2020). It will introduce you to practical applications of reinforcement learning in robotics while highlighting the complexities of training agents capable of generalizing to different environments and tasks. Your main objective is to develop and refine models capable of solving three distinct race types: Time Trial, Object Avoidance, and Head-to-Head. Each race type presents unique challenges, from optimizing speed on an open track to navigating around obstacles and competing against other agents. You will use reward function design, representation learning, hyperparameter tuning and incremental training methods to improve model performance across varied tracks and scenarios.

## 2 AWS DeepRacer

The AWS DeepRacer vehicle is a 1/18th scale autonomous racing car equipped with front-facing cameras for stereo vision, and LIDAR sensors, which enhance obstacle detection and depth perception. This sensor suite, combined with customizable reward functions, enables you to develop and test drive models that navigate real-world track scenarios using a deep reinforcement learning model learned from virtual training. For ease of use, in this project we provide a local simulation of the AWS DeepRacer environment together with a `deepracer_gym` package for a familiar `gymnasium` API via the `deepracer-v0` environment.


(a) Monocular camera (160 × 120, 8-bit, colored).

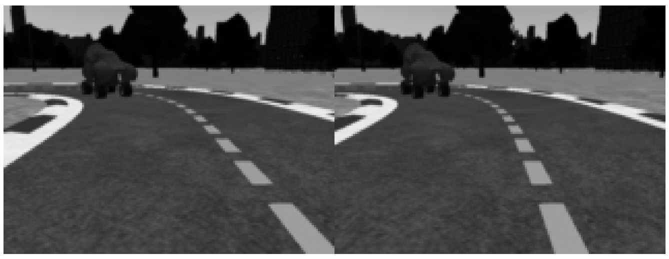

(b) Stereo cameras (each 160 × 120, 8-bit, greyscale).

Figure 1: Example input from different front-facing camera sensors.

## 2.1 State and Observation Spaces

At each time-step, a typical vehicle's state on a track *could* be represented as a tuple of state variables such as

$$(x, y, \theta, v, \dot{\theta}, p, s_{\text{progress}}),$$

where

- $x$ and $y$ represent the position of the DeepRacer vehicle on the track.
- $\theta$ is the heading angle of the vehicle, indicating its orientation relative to the track.
- $v$ is the current speed of the vehicle.
- $\dot{\theta}$ is the angular velocity, indicating the rate of change in the vehicle's heading.
- $p$ is the vehicle's distance from the centerline of the track.
- $s_{\text{progress}}$ is the cumulative progress percentage along the track.

However, these (or other state variables) may be difficult to measure or entirely inaccessible for a real vehicle on an arbitrary track and are therefore not included in the observation space. **Observations are only made via measurements from the sensors that a car is equipped with**, which include

- **Camera sensors:** You can choose one of two front-facing camera sensor settings – a color monocular camera or greyscale stereo cameras as shown in Fig. 1. At each time-step, we receive a measurement from the selected camera(s) as an image(s).
- **LIDAR sensor:** You can choose to include an optional LIDAR sensor that outputs a 64 dimensional vector representing ranges, i.e. radial distances (meters), to the surrounding obstacles. The 64 readings are uniformly spread over 360°, starting from $\sim -2.8°$ (to $\sim +2.8°$) counterclockwise. Note that LIDAR measurements beyond a minimum of 0.15m and a maximum of 1m are undefined.

Please consult the AWS DeepRacer developer guide for more details about different sensors and their possible use cases. For the purposes of this project, we shall restrict ourselves to the stereo cameras and LIDAR for all of the following experiments. These can be specified in the `configs/agent_params.json` file. Please see the `deepracer_gym` package documentation for instructions and examples on specifying sensors.

Generally, these sensor measurements are not sufficient to recover the state of the vehicle, and therefore the MDP is only partially observable. Nevertheless, some state variables together with additional auxiliary variables are accessible within the reward function to calculate the rewards. The same is also available in the auxiliary information returned by the `gymnasium.Env.step` function for debugging and evaluation purposes. For a complete set of these state and auxiliary variables, please see the AWS DeepRacer developer guide.

## 2.2 Action Space

The agent can drive the DeepRacer vehicle by adjusting the throttle to control the speed and adjusting the steering angle to control the direction. This can be defined in the `configs/agent_params.json` file via a discrete or continuous action space as follows.

- **Discrete:** The action space is a set of integers $a \in \{1, \cdots, n\}$ that enumerate a set of $n \in \mathbb{Z}^+$ pre-defined tuples/combinations of speed and steering angle.

- **Continuous:** The action space is a 2D vector representing speed and steering angle within set ranges.

See the `deepracer_gym` package documentation for instructions and examples on setting the action space.

## 2.3 Reward Function

The reward function must be defined in `configs/reward_function.py` in accordance with the task at hand. You can find an exhaustive list of the reward function parameters as well as reward function examples in the AWS DeepRacer developer guide. Note that these examples by themselves may not suffice for achieving the best possible performance (e.g., lap-time), but may serve as an inspiration for you to define your own reward function or make any modifications thereof.

## 2.4 Termination Conditions

The provided DeepRacer simulation terminates each episode when any of the following conditions are satisfied.

- **Lap completed:** When the agent finishes one lap around the track.
- **Crashed:** When the agent collides with an obstacle or a bot car.
- **Off track:** When the agent goes completely off of the track (i.e. none of the wheels are on the track).
- **Reversed:** When the agent goes in the opposite direction for 15 consecutive steps.

The first three are accessible in the reward function[1]. Furthermore, the simulation truncates the episode on the following conditions.

- **Immobilized:** When the agent moves less than 0.3 mm for 15 consecutive steps.
- **Time up:** When the agent reaches a maximum number of 100k steps in an episode.

## 2.5 Environment and Tasks

The environment, including the race track, race type and other relevant parameters can be defined in the `configs/environment_params.yaml` file. We shall discuss some of these configuration parameters below, but for details please see the DeepRacer-for-Cloud reference manual.

### 2.5.1 Race Types

You can simulate the following types of racing events by appropriately configuring the environment.

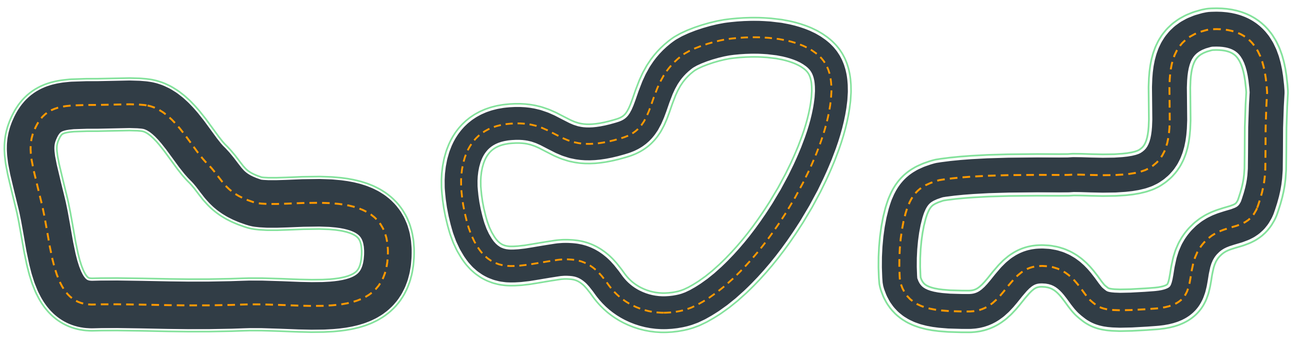
- **Time-Trial:** Race against the clock on an unobstructed track and aim to get the fastest lap time possible. Set `NUMBER_OF_OBSTACLES` to 0 and `NUMBER_OF_BOT_CARS` to 0.
- **Obstacle-Avoidance:** Race against the clock on a track with stationary obstacles and aim to get the fastest lap time possible. Set `NUMBER_OF_OBSTACLES` to 6 and `NUMBER_OF_BOT_CARS` to 0.
- **Head-to-Bot:** Race against one or more other vehicles on the same track and aim to cross the finish line while avoiding other vehicles. Set `NUMBER_OF_OBSTACLES` to 0 and `NUMBER_OF_BOT_CARS` to 3.

### 2.5.2 Evaluation Metrics

In this project we shall restrict ourselves to the race tracks in Fig. 2 by setting the `WORLD_NAME` variable accordingly. You are only allowed to use these for training and evaluation purposes. For any race type, the problem is considered solved when the agent can traverse 100% of the track (i.e., completes a lap) for 5 consecutive episodes across all 3 tracks. This is to ensure that the policy is robust across tracks and not just overfitted to the training track. We have provided a `src.utils.evaluate` function to make such an evaluation easier.

Additionally, we also provide calculation of lap-times for each evaluation run in `src.utils.evaluate`. These can be used as a tie-breaker to pick the best agent if multiple agents are able to solve the problem.

[1]Keep in mind that the `is_reversed` variable in the reward function is not the same as the above stated 'reversed' termination criteria.



(a) A to Z Speedway `reInvent2019_wide` (b) Smile Speedway `reInvent2019_track` (c) AWS Summit Raceway `Vegas_track`

Figure 2: Project race tracks (with their respective `WORLD_NAME` labels), listed in an increasing order of difficulty.

## 3 The Problem

This project is divided into two sequential parts, with each part building upon the previous one. In Part I, you will focus on developing an agent to solve the Time-Trial problem, optimizing its performance on specific tracks through custom reward functions and hyperparameter tuning. The best-performing models from Part I will then be used as the foundation for Part II, where you will apply further refinements and testing to achieve robust, competitive results across different race types.

### 3.1 Part I: Robust Time-Trial Across Tracks

In this part, your goal is to train an agent that can achieve the fastest possible lap times on the three specified tracks without any obstacles or competitor bots.

You will first design your solution to have an agent at least solve the simplest of the three project tracks. Once you have a working solution, you can decide to switch to the more challenging tracks if required.

Furthermore, since we can only pick one track at a time during training, you will need to find out what it takes to have a single agent solve all of the tracks.

#### 3.1.1 Solution Design

You will define and train your agent using the provided `deepracer_gym` environment and associated tools. This involves specifying several key components:

- **Action Space:** Define an action space as either discrete or continuous with appropriate parameterization.
- **Reward Function:** Design a custom reward function to solve the Time-Trial task. This is crucial for guiding the agent's learning.
- **RL Algorithm:** You may use any algorithm that you consider suitable for this problem.
- **Function Space:** Define the function space for the policy/value function approximation. This also includes any pre-processing you may perform on the observation samples in addition to the architecture of the neural-network, if used.
- **Hyper-Parameters:** Enumerate the key hyper-parameters for your implemented solution. Also specify the race track that you used for training your agent. Outline your training schedule if you choose to train the agent on multiple tracks separately or if you iteratively retrained your agent on all tracks. Try to choose a set of hyper-parameters that work independently of the choice of training track and/or schedule.

#### 3.1.2 Results and Analysis

**Training.** While training your Time-Trial agent(s):

- Choose any 3 metrics that you think are helpful to track during training. Plot them against the (global) number of steps executed by your algorithm.

- Analyze their trends and discuss whether the behavior aligns with your reward function's intent (e.g., is the agent's lap-time increasing or decreasing?).
- Discuss any challenges encountered during training (e.g., convergence issues, etc.) and how you addressed them.

**Evaluation.** After training and tuning your Time Trial agent(s)

- Use the provided `src.utils.evaluate` function for evaluation and write a commentary on the results.
- Visualize and discuss the agent's behavior using the provided `src.utils.demo` function. Make sure to save the video file in your repository.

## 3.2 Part II: Adaptation to Other Race Types

Building on your solution from Part I, this part explores adapting it for the more complex tasks of Object-Avoidance and Head-to-Bot race types. You may try any approach you think is appropriate, such as changing the components of your solution from Part I, using your Time-Trial agent to initialize your Object-Avoidance/Head-to-Bot agents to avoid complete re-training, or a combination thereof.

The main motivation is for you to learn about and experience the non-trivial and open-ended nature of most real-world RL tasks.

### 3.2.1 Adaptation, Additional Training and Adjustments

For each of the Object-Avoidance and Head-to-Bot race types, document and justify any changes you made to your solution from Section 3.1.1.

### 3.2.2 Results and Analysis

**Repeat** the training and evaluation analyses of Section 3.1.2 for **each** of the Object Avoidance and Head-to-Bot agents. This includes:

- **Training.** Select and plot 3 relevant metrics for each agent. Analyze their trends and highlight any training challenges encountered.
- **Evaluation.** Assess each agent's performance using the `src.utils.evaluate` function and provide a brief commentary. Record and save a demo video using `src.utils.demo`. Additionally, evaluate your trained Time-Trial agent from Part I on these new tasks to use as a performance baseline.

## 4 Implementation Details

Clone the project repository from https://github.gatech.edu/rldm/P4_deepracer and follow the detailed instructions in `SETUP.md` to prepare your development environment. In case you are unsuccessful in setting up your environment for whatever reason, please follow the instructions to use a PACE ICE machine instead.

Please make sure that your implementation complies with the following development guidelines.

- **Provided Code:** Familiarize yourself with the provided codebase.
  - `configs/`: Contains configuration files for the agent, environment, default hyperparameters, and the reward function. You will modify these extensively. Please make sure to save these for working solutions either by making a copy or logging via `tensorboard`.
  - `src/`: Contains the implementation for your solution in addition to the agent definition (`agents.py`), function approximation (`transforms.py`), and utility functions (`utils.py`).
  - `scripts/`: Includes scripts to start and stop the DeepRacer simulator.
  - `packages/deepracer_gym/`: Core wrapper exposing `gymnasium` API for the DeepRacer simulation.
- **Starting Point:** A dummy notebook `usage.ipynb` is provided to demonstrate basic interaction with the environment, policy visualization and evaluation.
- **Environment:** Keep the following in mind when using the `deepracer-v0` Gymnasium environment.

  - Not Vectorizable: Note that due to simulation constraints, the environment is **not vectorizable**.
  - Observation Handling: The `deepracer-v0` environment returns observations as a dictionary which is less straightforward to handle as opposed to just vectors. Use the provided `src.utils.make_environment` function instead which 'flattens' the observations into a vector. If you wish to 'un-flatten' the observations at any point, you may use the provided `src.transforms.UnflattenObservation` class.

- **Agent:** Make sure to inherit all of your agents from the `src.agents.Agent` class structure provided in `src/agents.py` to ensure compatibility with the utility functions.
- **Function Approximation:** You may use any function approximator to solve this problem, however we strongly recommend you use a non-linear function approximator such as a convolutional neural network (CNN). If you decide to train a neural network, you must train it using **PyTorch** as an automatic differentiation tool. Other neural network training libraries such as Tensorflow, Keras, Theano are not allowed. PyTorch is well liked amongst researchers for its pythonic feel. We have provided a default CNN encoder under `src.transforms` as an example. You may make any changes that you deem appropriate.
- **Workflow:** Use separate notebooks or scripts for Part I and Part II to keep your experiments organized. Use `tensorboard` to monitor training progress and save configurations/hyper-parameters.

## 5 Deliverables

### 5.1 Code

- We have provided you with some template files and utility functions to help you build your solution. Please make sure that it complies with the instructions in Section 4.
- Make sure to **delete the `.git` folder from the template files** before pushing your implementation to your personal Georgia Tech GitHub repository found here: `https://github.gatech.edu/gt-omscs-rldm`
  - The quality of the code is not graded. You don't have to spend countless hours adding comments, etc. But, it will be examined by the TAs.
  - Make sure to include a `README.md` in your repository with clear, detailed instructions for running your code—especially if your environment differs from the default setup.
  - The `README.md` file should be placed in the `project_4` folder in your repository.
  - If using a notebook (e.g., Jupyter), include a `.py` export along with it.
  - **Make sure to include** 3 **videos** in your `README.md` generated with the `src.utils.demo` function, one for each of Time-Trial, Object-Avoidance and Head-to-Bot race-types with the respective trained agent. Chose any track that you wish. **You will be penalized by 10 points if you do not include any videos,** 10/3 **points for each video**.
  - You will be penalized by 25 points if you do not have any code in your github repo or if you do not merge your code into the main branch.
  - Missing the git hash for your last commit in the paper will result in a 15-point penalty.
  - Using a Deep RL library instead of providing your own work will earn you a 0 grade on the project, and you will be reported for violating the Honor Code.

### 5.2 Report

- Write a paper describing your agent(s) and the experiments you ran.
  - Your report must use the **RLC 2024 Template**, available at this link. The report must not exceed **8 pages**, excluding citations.
  - Include the **git hash** for your last commit to the GitHub repository in the header or on the first page of your paper.
  - Make sure your graphs are legible and you cite sources properly.
  - Save the paper in PDF format.
  - Submit to **Gradescope**!

- Visualization requirements:
  - Graph: For each race type—Time-Trial (Part I), Object-Avoidance (Part II), and Head-to-Bot (Part II)—provide the following graphs along with discussion of results:
    - Evaluation plots showing progress and lap-time.
    - Training plots of 3 chosen metrics.
  - Any additional plots that support your findings and analysis.
- Narrative requirements:
  - Write a comprehensive report about your approach, experiments, and findings for Part I and Part II.
  - Explanation and justification of design choices made, modifications to reward functions, learning schedule, hyper-parameter experiments, etc. What worked best, what didn't, and what could have worked better.
    - Action Space
    - RL Algorithms
    - Reward functions
    - Function approximations
    - Hyper-parameters and their tuning
  - Reflect on how your simulation trained model might perform in a physical setting considering different lighting conditions, textures and sensor noise, obstacles, and other vehicles etc.
  - Description of pitfalls, challenges, and/or problems you encountered.
  - What would you try if you had more time?
  - Note: Even if you don't build an agent that solves the problem you can still write a solid paper.

## 5.3 Video

- Record a short video (maximum 5 minutes) explaining your approach to solving the Overcooked layouts, presenting your results, and highlighting the key lessons learned. Your video should follow a clear narrative:
  - Introduce the problem and describe the RL algorithm you selected, and why you selected it.
  - Explain your training process, design choices, and any challenges you faced.
  - Present your findings using visual aids (e.g., slides, plots, or a demo recording of your agent in action). You may reuse figures from your report, but do not scroll through the report itself.
  - Conclude with the most important lessons learned, including what you might do differently in the future.

  To protect your work, add a small watermark (e.g., your GT username) to any images or videos in your presentation. Do not include source code in the video.
- You may use Kaltura (provided by Georgia Tech), PowerPoint's video recording tool, or any other screen recording software to create your video. You are not required to appear on camera.
- Submit a link to your video hosted in a Georgia Tech enterprise service (Microsoft OneDrive, Box, Dropbox, or Media Space) along with the git hash of your final code in your report. The link must be set so that anyone within Georgia Tech can **view**, so that the instructional team can access and evaluate your submission.
- You will be penalized by 15 points if the video link is missing or the sharing settings do not permit the instructional team to view it.

### 5.4 Submission Details

**The due date is indicated on the Canvas page for this assignment.** Make sure you have set your timezone in Canvas to ensure the deadline is accurate.
Due Date: **Indicated as "Due" on Canvas**
Late Due Date [**20 point penalty per day**]: **Indicated as "Until" on Canvas**
The submission consists of:

- Your written report in PDF format (Make sure to include the git hash of your last commit).
- Your source code.

**To complete the assignment, submit your written report to Project 4 under CS7642 on Gradescope and submit your source code to your personal repository on Georgia Tech's private GitHub.**

You may submit the assignment as many times as you wish up to the due date, but, we will only consider your last submission for grading purposes. Late submissions will receive a cumulative 20 point penalty per day. That is, any projects submitted after midnight AOE on the due date will receive a 20 point penalty. Any projects submitted after midnight AOE the following day will receive another 20 point penalty (a 40 point penalty in total) and so on. No project will receive a score less than a zero no matter what the penalty. Any projects more than 4 days late and any missing submissions will receive a 0.

Please be aware, if Canvas marks your assignment as late, you will be penalized. This means one second late is treated the same as three hours late, and will receive the same penalty as described in the breakdown above. **Additionally, if you resubmit your project and your last submission is late, you will incur the penalty corresponding to the time of your last submission.** Submit early and often.

Finally, if you have received an exception from the Dean of Students for a personal or medical emergency we will consider accepting your project up to 7 days after the initial due date with no penalty. Students requiring more time should consider taking an incomplete for this semester as we will not be able to grade their project.

### 5.5 Grading and Regrading

When your assignments, projects, and exams are graded, you will receive feedback explaining your successes and errors in some level of detail. This feedback is for your benefit, both on this assignment and for future assignments. It is considered a part of your learning goals to internalize this feedback. This is one of many learning goals for this course, such as: understanding game theory, random variables, and noise. If you are convinced that your grade is in error in light of the feedback, you may request a regrade within a week of the grade and feedback being returned to you. A regrade request is only valid if it includes an explanation of where the grader made an error. **Create a private Ed Discussion post titled "[Request] Regrade Project 4"**. In the Details add sufficient explanation as to why you think the grader made a mistake. Be concrete and specific. We will not consider requests that do not follow these directions.

## 6 Resources

### 6.1 Lectures

- Lesson 8: Generalization.

### 6.2 Readings

- Chapter 9 (On-Policy Prediction with Approximation) of Sutton and Barto 2020.
- Chapter 13 (Policy Gradient Methods) of Sutton and Barto 2020.

### 6.3 Documentation

- **AWS DeepRacer developer guide:** Details on reward function, action spaces, and general concepts.
  - Main Page: `https://docs.aws.amazon.com/deepracer/latest/developerguide/index.html`
  - Reward Function Reference: `https://docs.aws.amazon.com/deepracer/latest/developerguide/deepracer-reward-function-reference.html`

- **DRfC:** https://aws-deepracer-community.github.io/deepracer-for-cloud/reference.html (for understanding environment parameters).
- `deepracer_gym` **Package:** See `packages/README.md` for usage of the local sim. environment wrapper.
- **Gymnasium:** https://gymnasium.farama.org/ (For understanding the RL environment API).
- **PyTorch:** https://pytorch.org/docs/stable/index.html
- **TensorBoard:** https://www.tensorflow.org/tensorboard/ (For visualizing training logs).
- **Convolutional Neural Networks (CNNs):** https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html (Image classification with CNNs in PyTorch).

## References

[Bal+19] Bharathan Balaji et al. *DeepRacer: Educational Autonomous Racing Platform for Experimentation with Sim2Real Reinforcement Learning*. 2019. arXiv: 1911.01562 [cs.LG]. URL: https://arxiv.org/abs/1911.01562.

[Bal+20] Bharathan Balaji et al. “DeepRacer: Autonomous Racing Platform for Experimentation with Sim2Real Reinforcement Learning”. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 2746–2754. DOI: 10.1109/ICRA40945.2020.9197465.

[SB20] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2nd Ed. MIT press, 2020. URL: http://incompleteideas.net/book/the-book-2nd.html.