Learn-to-Race

Jimmy Herman

Jan 16, 2022

INTRODUCTION:

1 Environment Overview **1**
1.1 Introduction . . . 1
1.2 Baseline Models . . . 2
1.3 Action Space . . . 2
1.4 Observation Space . . . 2
1.5 Customizable Sensor Configurations . . . 2
1.6 Interfaces and configuration . . . 3
1.7 Racetracks . . . 3
1.8 Research Citation . . . 3

2 Setup & Installation **5**
2.1 Racing Simulator . . . 5
2.2 Learn-to-Race Framework . . . 6
2.3 Runtime Steps . . . 6

3 Baselines **9**
3.1 Demonstration Against Human Experts . . . 9
3.2 Random Action Agent . . . 9
3.3 Soft Actor-Critic . . . 9
3.4 Model Predictive Control . . . 11

4 Task Overview **13**
4.1 Overview . . . 13
4.2 Metrics . . . 14
4.3 Legal Modifications . . . 15
4.4 Training Only . . . 15
4.5 Illegal Modifications . . . 15

5 Default Camera Configuration **17**
5.1 Overview . . . 17
5.2 Modifying Cameras . . . 17
5.3 Creating Cameras . . . 18

6 Multimodal **19**
6.1 Setting the Environment . . . 19
6.2 Environment Observations . . . 19

7 Racetracks & L2R Datasets **21**
7.1 Changing the Track . . . 21
7.2 L2R Dataset . . . 21
7.3 Racetrack File Format . . . 22

7.4 Basic Visualization of Tracks 22

8 Creating Custom Sensor Configurations **25**
8.1 Overview 25
8.2 Creating Sensors 25
8.3 Using Your New Sensor 26

9 Core Package **29**
9.1 l2r.core.controller 29
9.2 l2r.core.templates 29
9.3 l2r.core.tracker 30

10 Envs Package **31**
10.1 l2r.envs.env 31
10.2 l2r.envs.reward 31
10.3 l2r.envs.utils 31

11 Indices and tables **33**

Python Module Index **35**

Index **37**

CHAPTER

ONE

ENVIRONMENT OVERVIEW

1.1 Introduction

Learn-to-Race (L2R) is an *OpenAI-gym* compliant, multimodal control environment, where agents learn how to race.

Unlike many simplistic learning environments, ours is built around high-fidelity simulators, based on Unreal Engine 4, such as the Arrival Autonomous Racing Simulator—featuring full software-in-the-loop (SIL) and even hardware-in-the-loop (HIL) simulation capabilities. This simulator has played a key role in bringing autonomous racing technology to real life in the Roborace series, the world's first extreme competition of teams developing self-driving AI. The L2R framework is the official training environment for Carnegie Mellon University's Roborace team, the first North American team to join the international challenge.

Autonomous Racing poses a significant challenge for artificial intelligence, where agents must make accurate and high-risk control decisions in real-time, while operating autonomous systems near their physical limits. The L2R framework presents *objective-centric* tasks, rather than providing abstract rewards, and provides numerous quantitative metrics to measure the racing performance and trajectory quality of various agents.

As in the real-world, the Arrival Autonomous Racing Simulator is not time-invariant. In order to generate deployable solutions, latencies that result from model inference and from algorithm optimisation must be considered in the design of control policies. For example (model inference latency), whereas using large-capacity visual processing backbones may be desirable from a representation learning perspective, these large processing pipelines can induce significant inference latency for real-time applications. As another example (algorithm optimisation latency), the optimisation of learning-based approaches (e.g., reinforcement learning algorithms, with neural function approximators) often involves performing gradient steps, which, if performed in the middle of an episode, can also introduce latency. Latencies that go unaddressed could interfere with the control algorithm's ability to send quick control commands and, thus, could cause the vehicle to drive outside the admissible area or to engage in unsafe behaviour on the road. Both of these situations pose fascinating and exciting challenges for autonomous racing research. We hope to encourage work on these and other interesting problems, through the use of the Learn-to-Race framework.

For more information, please read a couple of our conference papers:

- Learn-to-Race: A Multimodal Control Environment for Autonomous Racing
- Safe Autonomous Racing via Approximate Reachability on Ego-vision

1.2 Baseline Models

We provide three baseline models, to demonstrate how to use L2R: a random agent, a model predictive control (MPC) agent, a Soft Actor-Critic reinforcement learning (RL) agent, and an imitation learning agent based on the MPC's demonstrations.

The RandomAgent executes actions, completely at random. The MPCAgent recursively plans trajectories according to a dynamics model of the vehicle, then executes actions according to the current plan. The SACAgent is a learning-based method, which relies on the optimisation of a stochastic policy, model-free.

1.3 Action Space

In the Learn-to-Race tasks, agents execute actions in the environment, according to steering and acceleration control, each supported by the simulator on a continuous range from -1.0 to 1.0.

Action	Type	Range
Steering	Continuous	[-1.0, 1.0]
Acceleration	Continuous	[-1.0, 1.0]

To provide additional flexibility for learning-based approaches, the L2R framework supports a scaled action space of [-1.0, 1.0] for steering control and [-16.0, 6.0] for acceleration control, by default. You can modify the boundaries of the action space by changing the parameters for *env_kwargs.action_if_kwargs* in *params-env.yaml*.

Negative acceleration commands perform braking actions, until the vehicle is stationary. If negative acceleration commands continue after the vehicle is stationary, the vehicle will reverse.

While you *can* change the gear, in practice we suggest forcing the agent to stay in drive since the others would not be advantageous in completing the tasks we present (we don't include it as a part of the action space). Note that negative acceleration values will brake the vehicle.

1.4 Observation Space

We offer two high-level settings for the observation space: vision-only and multimodal. In both, the agent receives RGB images from the vehicle's front-facing camera, examples below. In the latter, the environment also provides sensor data, including pose data from the vehicle's IMU sensor.

1.5 Customizable Sensor Configurations

One of the key features of this environment is the ability to create arbitrary configurations of vehicle sensors. This provides users a rich sandbox for multimodal, learning based approaches. The following sensors are supported and can be placed, if applicable, at any location relative to the vehicle:

- RGB cameras
- Depth cameras
- Ground truth segmentation cameras
- Fisheye cameras
- Ray trace LiDARs

- Depth 2D LiDARs
- Radars

Additionally, these sensors are parameterized and can be customized further; for example, cameras have modifiable image size, field-of-view, and exposure. Default sensor configurations are provided in *env_kwargs.cameras* and *sim_kwargs* in *params-env.yaml*. We provide further description on sensor configuration

You can create cameras anywhere relative to the vehicle, allowing unique points-of-view such as a birdseye perspective which we include in the vehicle configuration file.

For more information, see Creating Custom Sensor Configurations

Whereas we encourage the use of all sensors for training and experimentation, only the CameraFrontRGB camera will be used for official L2R task evaluation, e.g., in our Learn-to-Race Autonomous Racing Virtual Challenges.

1.6 Interfaces and configuration

The environment interacts with additional modules in the overall L2R framework, such as the racetrack mapping (for loading and configuring the world), the Controller (which interfaces with an underlying simulator or vehicle stack) and the Tracker (which tracks the vehicle state and measures progress along the racetrack).

Whereas each of these interfaces can be further configured from *params-env.yaml*, the default values provided will be used for official L2R task evaluation, e.g., in our Learn-to-Race Autonomous Racing Virtual Challenges.

- Tracker (l2r/core/tracker.py), configured via *env_kwargs* in *configs/params-env.yaml*
- Controller (l2r/core/controller.py), configured via *env_kwargs.controller_kwargs* in *configs/params-env.yaml*
- racetrack (l2r/racetracks/mapping.py), configured via *sim_kwargs* in *params-env.yaml*

1.7 Racetracks

We currently support two racetracks in our environment, both of which emulate real-world tracks. The first is the Thruxton Circuit, modeled off the track at the Thruxton Motorsport Centre in the United Kingdom. The second is the Anglessey National Circuit, located in Ty Croes, Anglesey, Wales.

Additional tracks are used for evaluation, e.g., in open Learn-to-Race Autonomous Racing Virtual Challenges, such as the Vegas North Road track, located at Las Vegas Motor Speedway in the United States.

We will continue to add more racetracks in the future, for both training an evaluation.

1.8 Research Citation

Please cite this work if you use L2R as a part of your research.

```
@inproceedings{herman2021learn,
          title={Learn-to-Race: A Multimodal Control Environment for Autonomous Racing}
↪,
          author={Herman, James and Francis, Jonathan and Ganju, Siddha and Chen,␣
↪Bingqing and Koul, Anirudh and Gupta, Abhinav and Skabelkin, Alexey and Zhukov, Ivan␣
↪and Kumskoy, Max and Nyberg, Eric},
          booktitle={Proceedings of the IEEE/CVF International Conference on Computer␣
↪Vision},
```

(continues on next page)

(continued from previous page)

```
        pages={9793--9802},
        year={2021}
      }
```

CHAPTER TWO

SETUP & INSTALLATION

2.1 Racing Simulator

To use the Learn-to-Race environment, you must first request access, by filling out and returning a signed academic-use license.

Our environment interfaces with the Arrival Autonomous Racing Simulator via a SimulatorController object which can launch, restart, and control the simulator.

2.1.1 Simulator Requirements

Operating System: The racing simulator has been tested on Ubuntu Linux 18.04 OS.

Graphics Hardware: The simulator has been tested to run smoothly on NVIDIA GeForce GTX 970 graphics cards. The simulator has been additionally tested on the following cards:

- NVIDIA GeForce GTX 1070
- NVIDIA GeForce GTX 1080, 1080 Ti
- NVIDIA GeForce GTX 2080, 2080 Ti
- NVIDIA GeForce GTX 3080, 3080 Ti
- NVIDIA GeForce GTX 3090

Software Dependencies:

- Please install the appropriate CUDA and NVIDIA drivers.
- Please additionally install the following software dependencies:

```
$ sudo apt-get install libhdf5-dev libglib2.0-dev libglib2.0-dev ffmpeg libsm6 libxext6␣
↪apt-transport-https
```

2.1.2 Running the Simulator

After the signed academic-use license is returned and approved, you will be given the opportunity to download the Arrival Autonomous Racing Simulator (*.tar.gz file). The simulator is currently being distributed as part of the Learn-to-Race Autonomous Racing Virtual Challenge, with a base file footprint of 2.8 GB.

Open a temrinal screen and untar the simulator source, to a location of your choice:

```
$ cd /path/to/simulator/download/location
$ tar -xvzf /path/to/simulator/ArrivalSim-linux-{VERSION}.tar.gz
$ chmod -R 777 /path/to/simulator/ArrivalSim-linux-{VERSION}/
```

We recommend running the simulator as a dedicated Python process, by executing:

```
$ bash /path/to/simulator/ArrivalSim-linux-0.7.0.182276/LinuxNoEditor/ArrivalSim.sh -
↪openGL
```

Note: Users may receive a pop-up window, warning about OpenGL being deprecated in favour of Vulkan. It is safe to click 'Ok', to continue initialisation and use of the simulator.

2.2 Learn-to-Race Framework

2.2.1 Installation

Simply download the source code from the Github repository.

We recommend using a python virtual environment, such as Anaconda. Please download the appropriate version for your system. We have tested Learn-to-Race with Python versions 3.6 and 3.7.

Create a new conda environment, activate it, then install the Learn-to-Race python package dependencies:

```
$ conda env create -n l2r -m python=3.6        # create virtual environment
$ conda activate l2r                           # activate the environment
(l2r) $ cd /path/to/repository/
(l2r) $ pip install -r requirements.txt
```

2.3 Runtime Steps

1. Start the simulator (e.g., in a separate terminal window), if it has not already been started:

```
$ bash /path/to/simulator/ArrivalSim-linux-0.7.0.182276/LinuxNoEditor/ArrivalSim.sh -
↪openGL
```

2. Run/train/evaluate an agent, using the Learn-to-Race framework (e.g., within a *tmux* window):

```
$ cd /path/to/repository
$ cd l2r
$ tmux new -s development
$ conda activate l2r
(l2r) $ chmod +x run.bash
(l2r) $ ./run.bash -b random
```

2.3.1 Basic Agent Example (Random Agent)

Here is an example of an agent that chooses random actions from the action space, provided by the environment.

We provide such an agent called a `RandomAgent` with the source code below:

```
from core.templates import AbstractAgent
from envs.env import RacingEnv

class RandomActionAgent(AbstractAgent):
    """Reinforcement learning agent that simply chooses random actions.

    :param dict training_kwargs: training keyword arguments
    """
    def __init__(self, training_kwargs):
        self.num_episodes = training_kwargs['num_episodes']

    def race(self):
        """Demonstrative training method.
        """
        for e in range(self.num_episodes):
            print(f'Episode {e+1} of {self.num_episodes}')
            ep_reward = 0
            state, done = self.env.reset(), False

            while not done:
                action = self.select_action()
                state, reward, done, info = self.env.step(action)
                ep_reward += reward

        print(f'Completed episode with total reward: {ep_reward}')
        print(f'Episode info: {info}\n')

    def select_action(self):
        """Select a random action from the action space.

        :return: random action to take
        :rtype: numpy array
        """
        return self.env.action_space.sample()

    def create_env(self, env_kwargs, sim_kwargs):
        """Instantiate a racing environment

        :param dict env_kwargs: environment keyword arguments
        :param dict sim_kwargs: simulator setting keyword arguments
        """
        self.env = RacingEnv(
            max_timesteps=env_kwargs['max_timesteps'],
            obs_delay=env_kwargs['obs_delay'],
            not_moving_timeout=env_kwargs['not_moving_timeout'],
            controller_kwargs=env_kwargs['controller_kwargs'],
            reward_pol=env_kwargs['reward_pol'],
            reward_kwargs=env_kwargs['reward_kwargs'],
```

(continues on next page)

(continued from previous page)

```
        action_if_kwargs=env_kwargs['action_if_kwargs'],
        pose_if_kwargs=env_kwargs['pose_if_kwargs'],
        cameras=env_kwargs['cameras']
    )

    self.env.make(
        level=sim_kwargs['racetrack'],
        multimodal=env_kwargs['multimodal'],
        driver_params=sim_kwargs['driver_params']
    )

    print(f'Environment created with observation space: ')
    for k, v in self.env.observation_space.spaces.items():
        print(f'\t{k}: {v}')
```

Run the random agent baseline model

For convenience, we have provided a number of files to assist with training a model. To run the random agent baseline, you can simply run the script in the top level of the repository with the baseline flag `-b` with argument `random`:

```
$ chmod +x run.bash  # make our script executable
$ ./run.bash -b random
```

The agent will begin randomly taking actions in the environment and will print the reward for each episode upon completion.

Convenience Scripts

`run.bash` simply passes parameters files to Python scripts. The baseline configuration files contains a variety of parameters including:

1. training parameters
2. environment parameters (for the RL environment)
3. simulator parameters (for the simulator)

We recommend using this structure, or following a similar practice, to train models with the environment and keep track of different training runs.

CHAPTER

THREE

BASELINES

3.1 Demonstration Against Human Experts

3.2 Random Action Agent

Our RandomActionAgent is a basic demonstration of an agent interacting with the environment which we explain in the Getting Started section of the docs. This agent simply takes random actions in the environment with an action space that is restricted to only take non-negative acceleration values.

3.2.1 Usage

As mentioned previously, you need to have the docker image with the simulator to use the environment. Simply add the -b flag and argument random as a command line argument to ./run.bash to use this agent.

```
$ chmod +x run.bash  # make our script executable
$ ./run.bash -b random
```

3.3 Soft Actor-Critic

We also provide a more detailed demonstration of how to use the environment with our Soft Actor-Critic agent using OpenAI's Spinning Up Pytorch implementation with minor adjustments. Specifically these adjustments include wrapping methods which returned observations from the environment to first encode the raw images into a latent representation, waiting until the end of the episode to make gradient updates, and removing unused functionality.

3.3.1 Training Performance

For both tracks, we provide our agent's model after 1000 episodes which was slightly less than 1 million environment steps for the Las Vegas Track and slightly more for the Thruxton track.

3.3.2 Evaluation Performance

The SAC agent struggles when transferring its learned experience from the Thruxton track to the Las Vegas evaluation track even after 60 minutes of exploration as it learns to simply stop in the middle of the track to avoid the penalty of going out-of-bounds.

3.3.3 Usage

To run the trained model, simply provide `-b` flag and argument `sac` to `run.bash`. Both the encoder and checkpoint models were trained separately for each track, so if you would like to switch to the Thruxton track, be sure to change the encoder and checkpoint paths in `configs/params_sac.yaml` in addition to the track name.

```
$ chmod +x run.bash  # make our script executable
$ ./run.bash -b sac
```

3.3.4 Vision-Only Perception & Control

This agent learns non-trivial control of the race car exlusively from visual features. First, we pretrained a variational autoencoder on the provided sample image datasets to allow our agent to learn from a low-dimensional representation of the images. Our VAE is a slight modification of Shubham Chandel's implementation.

3.3.5 Restriction of the Action Space

For this agent, we restricted the scaled action space to `[-0.1, 4.0]` for acceleration and `[-0.3, 0.3]` for steering to allow for faster convergence.

3.3.6 Custom Reward Policy

Additionally, we modified the default reward policy for the environment to include bonus if the agent is near the center of the track for each step in the environment but only if it had made progress down the track. Doing so has numerous consequences including:

- encouraging the agent to safely stay near the middle of the track
- disincentivizing the agent from engaging in corner cutting
- implicitly rewarding the agent to drive more slowly

As such, this reward allows for faster convergence in terms of number of episodes before completing its first lap in the environment. However, we noticed that the agent learns to zig-zag; we believe this may be an intentional effort to slow down and gather more near-center bonuses.

3.4 Model Predictive Control

We include a model predictive control (MPC), non-learning agent with the environment too. This reference implementation demonstrates a controller which attempts to minimize tracking error with respect to the centerline of the racetrack at a pre-specified reference speed.

3.4.1 Performance

The MPC agent does well, completing laps consistently, on the Thruxton track by following a conservative trajectory. On the LVMS track, however, it seems to occasionally falter on the highest curvature points of the track.

3.4.2 Usage

To run the trained model, simply provide the `-b` flag and argument `mpc` to `run.bash`. Do note, however, that the MPC requires `torch<=1.4` unlike the SAC baseline.

```
$ chmod +x run.bash  # make our script executable
$ ./run.bash -b mpc
```

CHAPTER FOUR

TASK OVERVIEW

4.1 Overview

The task is to learn how to race using either vision-only or multiple input modalities. While Arrival's racing simulator and the L2R framework is suitable for a wide variety of approaches including:

- classical control
- pre-planning trajectories
- reinforcement learning (RL)
- imiatation learning (IL)

We are most interested in agents which learn how to perceive their surroundings and effectively control the race car. We are also familiar with the success of RL approaches and fully expect, and encourage, the community to create agents which are significantly better than our human benchmarks. However, learning-based approaches often have poor sample efficiency and fail at generalizing to new scenarios. Humans, on the other hand, are good at quickly adapting to new situations, such as racing on a new, unseen track after only a short warmup period. The L2R task will challenges agents' ability race in such a way.

4.1.1 Task Definition

Rather than giving agents an unbounded amount of time to perfectly overfit to a racetrack, the L2R task allows only a limited look at a new track, much like a Formula 1 driver gets only a brief practice session prior to the actual race. More concretely, agent assessment involves two stages:

(1) **Pre-evaluation:** agents will have access to an unseen, evaluation racetrack for 60 minutes with unfrozen weights. The agent is free to use this time for any purpose they deem necessary, but we generally expect agents to transfer their prior racing knowledge to the new environment after a brief exploration period.

(2) **Evaluation:** to qualify for evaluation, the agent must demonstrate that it can complete at least 1 lap during the pre-evaluation stage, subject to a modest maximum time limit. Successful agents will be evaluated on the testing racetrack and their metrics, defined below, will be recorded and updated on a leaderboard. To prevent competitors from unfairly learning on the test track, submissions will only be able to see their agent's results and will not have access to any model updates during the pre-evaluation stage.

Note: We currently only support single vehicle racing, but hope to introduce a multi-agent environment in the future.

4.1.2 Input Modalities

We present two distinct sets of information available to agents. All information available to agents use virtual sensors that emulate their real counterparts, so the agent does not have access to any privileged information. A separate leadboard will be used for agent's using the more restricted *vision-only* input mode.

Vision-Only *The agent only has access to raw pixel values from the vehicle's cameras*

Multimodal *In addition to the cameras, we provide the agent with sensor data, primarily from the vehicle's IMU sensor*

4.2 Metrics

Learn-to-Race defines numerous metrics for the assessment of an agent's performance listed below. These are provided to agents upon episode termination in the *info* of the last environment step.

Metric	Definition
Episode Completion Percentage	Percentage of the 3-lap episode completed
Episode Duration	Duration of the episode, in seconds
Average Adjusted Track Speed	Average speed, adjusted for environmental conditions, in km/h
Average Displacement Error	Euclidean displacement from the track centerline, in meters
Trajectory Admissibility	Measurement of the safety of the trajectory
Trajectory Efficiency	Ratio of track curvature to trajectory curvature
Movement Smoothness	Log dimensionless jerk based on accelerometer data

4.2.1 Basic Metrics

A successful episode is defined as completing 3 laps, from a standing start, without 2 wheels going out-of-bounds at any point in time (1 is permissable, but considered *unsafe*). L2R provides basic metrics like the percentage of the 3 laps completed, the lap times for each successfully completed lap, the total time of the episode, and the average speed of the vehicle.

4.2.2 Trajectory Quality

To understand the quality of an agent's trajectories, L2R also includes metrics like *average displacement error* which is simply the agent's average distance from the centerline which is particularly useful measuring a controllers ability to stay near its target. We also include *trajectory admissibility* or α, shown below, where t_u is the cumulative time spent unsafely with 1 wheel out-of-bounds and t_e is the total length of the episode.

$$\alpha = 1 - \sqrt{\frac{t_{\mathrm{u}}}{t_e}}$$

A perfectly admissable trajectory is 1.0 with no time spent outside of the drivable area. Furthermore, we provide a *trajectory efficiency* ratio which is the ratio of curvature of the racetrack's centerline to the curvature of the trajectory, measured parametrically using the root mean square. Strong racing agents should minimize their curvature to maintain high speeds, for example, by cutting corners, and have an efficiency of at least 1.0.

Warning: If the agent doesn't complete the entire episode, the trajectory efficiency metric will likely be distorted since it would be comparing a partial trajectory, which may exclude high curvature areas of the track, to the entire racetrack.

Good racing agents should also be able to anticipate the need for changes in velocity and have the ability to smoothly control such changes. L2R also includes a *movement smoothness* measure, the negated log dimensionless jerk, η_{ldj}, which quantifies the smoothness of the agent's acceleration profile.

$$\eta_{ldj} = \ln \left(\frac{(t_2 - t_1)^3}{v_{peak}^2} \int_{t_1}^{t_2} \left| \frac{d^2 v}{dt^2} \right|^2 dt \right)$$

Agents that tend to jerk the vehicle or brake violently, both dangerous maneuvers, will have a worse movement smoothness measure.

4.3 Legal Modifications

For the purpose of benchmarking, we require that you adhere to some degree of requirements. There are no restrictions in the modification or usage of:

- exploration or learning method
- incentive method (reward function)
- network architecture
- pre-trained perception models
- the delay between action and observation
- changing the action space

4.4 Training Only

Certain camera settings must be considered training-only if they are realistically accessible to a physical racecar. The following camera settings are not available during evaluation:

- Segmentation cameras
- Cameras not touching the vehicle (for example, birdseye views)

4.5 Illegal Modifications

- Not using the default vehicle in the simulator (DevBot 2.0)
- Changing any physical parameters of the simulator such as the friction settings; we are *not* concerned about sim2real transfer
- Modifying the tracker method that would influence termination conditions or lap timing

CHAPTER

FIVE

DEFAULT CAMERA CONFIGURATION

5.1 Overview

The agent has access to camera sensor data which are raw pixel values (with bounds of 0 and 255). The default vehicle configuration includes 8 cameras with names:

- CameraFrontRGB (CameraFrontSegm)
- CameraLeftRGB (CameraLeftSegm)
- CameraRightRGB (CameraRightSegm)
- CameraBirdsEyeRGB (CameraBirdsEyeSegm)

To include any subset of these cameras, simply include the camera's name and parameters under `env_kwargs.cameras`. For example:

```
env_kwargs:
  cameras:
    CameraFrontSegm:
      Addr: 'tcp://0.0.0.0:9008'
      Format: SegmBGR8
      Width: 512
      Height: 384
      bAutoAdvertise: True
```

5.2 Modifying Cameras

The camera is flexible in terms of both the field-of-view angle, dimensions of the images, type of camera, and position. To modify the camera, simply change the parameters:

```
GenericCamera:                # Camera name which will be a key of observation␣
↪dictionary
  Addr: 'tcp://0.0.0.0:9008'  # Address camera will publish to
  Format: ColorBGR8           # ColorBGR8, SegmBGR8, HdrBGR8
  FOVAngle: 120               # modifiable field of view parameter, in degrees
  Width: 256                  # modifiable image width parameter, in number of pixels
  Height: 256                 # modifiable image height parameter, in number of pixels
  bAutoAdvertise: True
```

To set the environment to vision-only, set the `multimodal` parameter to `False` in the `configs/params.yaml` file.

```
env_kwargs:
  multimodal: False
  ...
```

5.3 Creating Cameras

See Creating Custom Sensor Configurations

CHAPTER

SIX

MULTIMODAL

6.1 Setting the Environment

The multimodal option provides visual features to the agent in an identical manner to the visual only feature set, but it also includes data from the vehicle's IMU sensor along with a few other pieces of data. We expect that the performance of agents with multimodal sensory data to be better than that of visual only agents. To set the environment to multimodal, simply modify the `multimodal` parameter to `True` in the `configs/params.yaml` file:

```
env_kwargs:
  multimodal: True       # when True, both images and pose data are provided to agent
  max_timesteps: 5000
  ...
```

6.2 Environment Observations

Setting this parameter to `True` will change the return type of the `step()` method of the RacingEnv class to return a `spaces.Dict` containing:

Track ID a numeric identifier of the current track, relevant for multi-track training

Camera Images a numpy array of shape **(image_width, image_height, 3)**

Additional Data a numpy array of shape (30,) with the following data:

Array Indicies	Data
0	steering request
1	gear request
2	mode
3,4,5	direction velocity in m/s
6,7,8	directional acceleration in m/s^2
9,10,11	directional angular velocity
12,13,14	vehicle yaw, pitch, and roll, respectively
15,16,17	center of vehicle coordinates in the format (y, x, z)
18,19,20,21	wheel revolutions per minute (per wheel)
22,23,24,25	wheel braking (per wheel)
26,27,28,29	wheel torque (per wheel)

CHAPTER SEVEN

RACETRACKS & L2R DATASETS

7.1 Changing the Track

Our environment uses three racetracks in the racing simulator, all of which are modeled off of real-world racetracks. The first is the North Road track at Las Vegas Motor Speedway located in the United States. The second is modeled off of the Thruxton Circuit track located in the United Kingdom. The third is Anglesey National based off the track at the Anglesey Circuit.

> **Warning:** The LVMS track is currently used as an evaluation track, so its map will remain unavailable.

In the parameters file, you can specify the track you would like to use in the `params.yaml` file located in `sim_kwargs` as follows:

```
sim_kwargs:
        racetrack: ['Thruxton', 'AngleseyNational']  # 'VegasNorthRoad' is used for␣
↪evaluation
        ...
```

If more than one track is specified, the environment will randomly select a track at the beginning of each episode by default. Furthermore, the beginning of each episode will be a randomly selected point on the track unless the environment is in evaluation mode.

```
env.eval()  # start episodes at the finish line
env.training()  # start episodes at a random location
```

7.2 L2R Dataset

Some users may wish to pre-train models, such as an encoder or segmentation model, on a sample of images from the vehicle's camera. While you can collect your own data from the environment, we provide datasets of each training track for such purposes. The Thruxton dataset includes 10,600 complete transitions including the sensor data, camera image, and action executed by the *MPC agent <baselines.html#Model-Predictive-Control>*, so it can also be used for immitation based approaches. This data was collected across 9 full laps around the Thruxton circuit track.

Images were saved using `` `numpy.savez_compressed()` `` with arrays `pose_data` and `image`. The images are in RGB format with a width of 512 pixels, height of 384 pixels, and a field of view of 90 degrees. Loading the images into a numpy array can be done as follows:

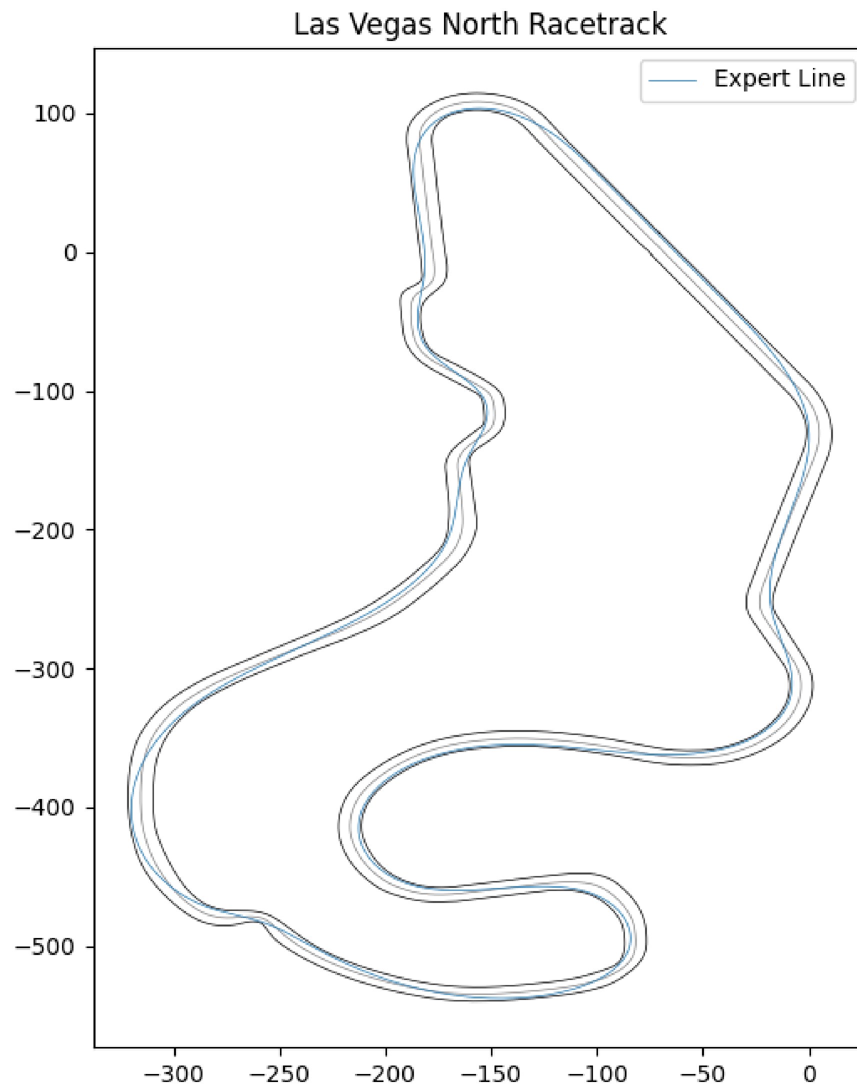
```
import matplotlib.pyplot as plt
import numpy as np

# load transition 100
filename = './transitions_100'
transition = np.load(filename + '.npz')

# view files, output is: ['image', 'multimodal_data', 'action']
print(loaded.files)

# pose_data is of shape (30,), image (384, 512, 3), and action (2,)
sensor_data, image, action = transition['multimodal_data'], transition['image'],␣
↪transition['action']

# save the image
plt.imsave(f'{filename}.png', transition['image'])
```

7.3 Racetrack File Format

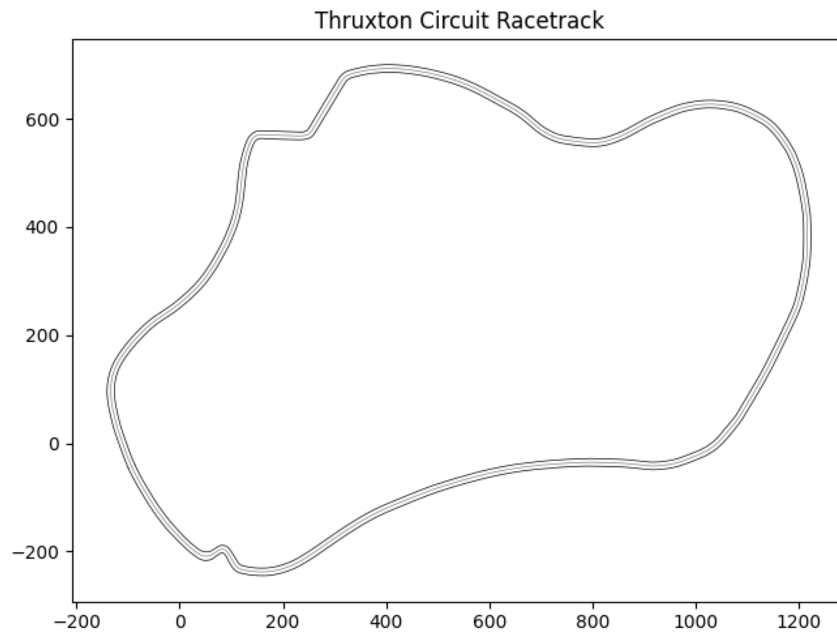
The racetrack files are in JSON format. Each contains coordinates of the inside, outside, and center lines of the track in East, North, Up convention. The ENU coordinate system is a *local* coordinate system that requires a reference point, which is also included in the racetrack files. Our RL environment converts the coordinates of the vehicle to ENU before returning them from the `step()` method.

7.4 Basic Visualization of Tracks

VegasNorthRoad has an expert trajectory included which some users may find useful for immitation learning approaches. Users are certainly welcome to generate their own expert trajectories.



The Thruxton Circuit track is known for its high speeds which will present unique challenges to drivers.



The following figure is from the Anglesey Circuit website highlighting the highly technical features of this exciting track.

CHAPTER

EIGHT

CREATING CUSTOM SENSOR CONFIGURATIONS

8.1 Overview

You can create arbitrary configurations of vehicle sensors with our environment. The following sensors are supported and can be placed, if applicable, at any location relative to the vehicle:

- RGB cameras
- Depth cameras
- Ground truth segmentation cameras
- Fisheye cameras
- Ray trace LiDARs
- Depth 2D LiDARs
- Radars

We will go over a brief tutorial on how to create and configure a custom camera using our environment.

Note: The L2R SimulatorController cannot create new sensors in the simulator, but it can enable sensors and modify their configuration.

8.2 Creating Sensors

By default, the simulator will load the previously saved vehicle configuration.

8.2.1 (Option 1) User Interface

1. Run the simulator and select an arbitrary map
2. On the right panel, select “Vehicle Sensor Settings”
3. In the top right, select “Add Sensor”
4. Select the sensor you wish to create
5. Configure the sensor to your choosing
6. Press “ESC” then select “Save All” or “Save Vehicle” in the top panel

Optionally, you can export the vehicle configuration by selecting "Vehicle Settings" in the right panel, scrolling down, and exports to a JSON or Yaml file.

8.2.2 (Option 2) Configuration File

Our repository contains a default vehicle configuration file which serves as a valuable reference tool.

1. Simply modify parameters or add new sensors as you see fit
2. Add your file to the appropriate simulator directory, for example, `ArrivalSim-linux-0.7.0.182276/LinuxNoEditor/Engine/Binaries/Linux/l2r_vehicle_conf.yaml`
3. Run the simulator and select an arbitrary map
4. Select "Vehicle Settings" in the right panel
5. Scroll down, and load your updated configuration
6. Press "ESC" then select "Save All" or "Save Vehicle" in the top panel

For example, we can add a rear facing camera named "CameraRearRGB" by appending this camera configuration to l2r_vehicle_conf.yaml then following the steps above.

```
version: 1.1
cameras
  - name: CameraRearRGB
    enabled: false
    model: ARRIVAL Generic Camera
    pose:
      x: -1.000  # 1m behind the reference point, slightly outside the vehicle
      y: 0.000
      z: 0.500   # 50cm above the reference point
      pitch: 0.0
      roll: 0.0
      yaw: 180.0  # rotate the camera around the Z-axis by 180 degrees
    transport:
      zmq: tcp://0.0.0.0:9999
```

8.3 Using Your New Sensor

To use your newly created camera, you simply need to add it to your parameter configuration file. The random action baseline configuration serves as a good reference. Once added, the observation returned from the environment's `step()` method will include the new images which are accessible via the camera's name. For example:

```
env_kwargs:
  cameras:
    CameraRearRGB:
      Addr: 'tcp://0.0.0.0:8008'  # make sure this address is unique
      Format: ColorBGR8
      FOVAngle: 90
      Width: 512
      Height: 384
      bAutoAdvertise: True
```

Rear facing images will be available like:

```
while not done:
    action = self.select_action()
    state, reward, done, info = self.env.step(action)
    rear_image = state['CameraRearRGB'] # numpy array
```

CHAPTER NINE

CORE PACKAGE

9.1 l2r.core.controller

9.2 l2r.core.templates

Note: The templates module provides useful abstract classes which we recommend using for compatability with the RacingEnv.

class l2r.core.templates.**AbstractAgent**(**args*, ***kwargs*)

Bases: `abc.ABC`

Abstract agent class. A potentially useful template for racing agents.

abstract select_action()

Select an action to take.

class l2r.core.templates.**AbstractInterface**(**args*, ***kwargs*)

Bases: `abc.ABC`

Abstract simulator interface to receive data from the simulator.

abstract get_data()

This method is used to return the most up-to-date information from the interface.

abstract reset()

Used to reset the interface, often to clear existing data.

abstract start()

The start method is used to start communication with the simulator.

class l2r.core.templates.**AbstractReward**(**args*, ***kwargs*)

Bases: `abc.ABC`

Abstract reward class. It is recommended that new reward policies follow this template so that they are compatible with the RacingEnv.

abstract get_reward(*state*, ***kwargs*)

Return the reward for the provided state.

Parameters **state** (*varies*) – the current environment state

abstract reset()

Reset the reward policy.

`set_track`(*inside_path*, *outside_path*, *centre_path*, *car_dims*)

Store the track and vehicle information as class variables. This is useful for evaluating the reward based on the position of the vehicle.

Parameters

- **inside_path** (*matplotlib.Path*) – ENU coordinates of the inside track boundary
- **outside_path** (*matplotlib.Path*) – ENU coordinates of the outside track boundary
- **centre_path** (*matplotlib.Path*) – ENU coordinates of the track's centerline
- **car_dims** (*list*) – dimensions of the vehicle in meters: [length, width]

9.3 l2r.core.tracker

CHAPTER

TEN

ENVS PACKAGE

10.1 l2r.envs.env

10.2 l2r.envs.reward

10.3 l2r.envs.utils

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

l

l2r.core.templates, 29

INDEX

A

AbstractAgent (*class in l2r.core.templates*), 29
AbstractInterface (*class in l2r.core.templates*), 29
AbstractReward (*class in l2r.core.templates*), 29

G

get_data() (*l2r.core.templates.AbstractInterface method*), 29
get_reward() (*l2r.core.templates.AbstractReward method*), 29

L

l2r.core.templates
    module, 29

M

module
    l2r.core.templates, 29

R

reset() (*l2r.core.templates.AbstractInterface method*), 29
reset() (*l2r.core.templates.AbstractReward method*), 29

S

select_action() (*l2r.core.templates.AbstractAgent method*), 29
set_track() (*l2r.core.templates.AbstractReward method*), 29
start() (*l2r.core.templates.AbstractInterface method*), 29