RallyAI

Sheila, Sayo and Brandon

Aug 21, 2020

CONTENTS

POTENTIAL MOTORS

CHAPTER

ONE

SETUP RALLYAI

1.1 Installation

Download the following repositories to the same parent folder:

- /Potential-Motors/RallyAI_v2
- /Potential-Motors/BasicSimulation
- /Potential-Motors/RallyAI_datasets

Navigate to `/RallyAI_v2/.`. These commands will install all necessary libraries and register the gym environment, respectively.

```
pip install -r requirements.txt
pip install -e .
```

We will make note that as of August 11th, 2020 we are using the bicycle simulation (python), so the following commands are unneccesary. To build the shared library for the simulation, navigate to `BasicSimulation/C/` and run

```
make -f MakeShared
sudo make -f MakeShared install
```

Note: You may also need to run `apt-get install -y libsm6 libxext6 libxrender-dev`

1.2 Configuration

Now that all necessary packages are installed, we will outline how to configure and tune your experiments.

The configuration file is located at `RallyAI_v2/RL_Training/config/simple_train.json`. Inside the file you will see a dictionary of dictionaries, defining multiple variables which will affect the training. The tables below are different sections from this config file.

Param	Type	Range	Description
timesteps	int	(0,inf)	Total number of timesteps for this training session
num_agents	int	(0,inf)	The number of agents running in parallel
randomize	bool	N/A	Defining if randomization is used for user states
run_on_aws	bool	N/A	Defines if we are running on AWS
algorithm	str	N/A	Determines what algorithm to use (stablebaselines)
load_model_path	str	N/A	Location and name of zip file to load weights from

The 'Network structure' table defines the architecture of the nueral network. Note that the param 'architecture' is a list which defines the dimension and number of hidden layers only.

Param (Network structure)	Type	Range	Description
policy_type	str	N/A	Type of model type (i.e. "fcn", "lstm", etc)
architecture	list, int	(0,inf)	Structure of policy (hidden) layers
lstm		N/A	
policy_kwargs		N/A	

The 'Input source' table contains parameters regarding how the simulation/vehicle are controlled. The 'input_file' is the style of the file used to for controls. For example, "constant" will select a constant value for the user inputs.

Param (Input source)	Type	Range	Description
input_file	str	N/A	
episode_length	int	(0,inf)	Number of steps or length of input file
vehicle	str	'Newton' or 'GWagon'	Vehicle parameters used in simulation

The WandB parameters are used for logging information to the Weights & Biases services.

Param (WandB)	Type	Range	Description
wandb_params/log_to_wandb	bool	N/A	Defines if the training data will be logged to W&B
wandb_params/log_frequency	int	(0,inf)	How frequently the data is sent to W&B
wandb_params/run_name	str	N/A	Name used for W&B run, also for saving weights
wandb_params/proj_name	str	N/A	Name used for W&B project, also for saving weights
wandb_params/time_code	bool	N/A	Defines wether or not to time sections of code

The below parameter directly relate with the algorithm that is chosen for training. Since we mostly use PPO, those params is what we have listed below. Additionally, the ranges given below are also only applicable for PPO. A more detailed explanation of the hyperparameters can be found here.

Param (Hyperparameters)	Type	Range	Description
hyperparams/learning_rate_start	float	[0.003, 5e-6]	Initial learning rate
hyperparams/learning_rate_end	float	[0.003, 5e-6]	Final learning rate (less than or equal to init lr)
hyperparams/gamma	float	[0.8,0.9997]	Discount Factor Gamma
hyperparams/vf_coef	float	[0.5,1]	Value Function Coefficient Range
hyperparams/ent_coef	float	[0,0.01]	Entropy Coefficient
hyperparams/lam	float	[0.9,1]	GAE Parameter Lambda
hyperparams/cliprange	float	{0.1,0.2,0.3}	Clipping
hyperparams/noptepochs	int	[3,30]	Epochs
hyperparams/nminibatches	int	[4,4096]	Minibatch
hyperparams/n_steps	int	[32,5000]	Horizon
hyperparams/max_grad_norm	float		
hyperparams/verbose	int	{0,1,2}	Amount of detail in algorithm output

Lastly, the reward parameters are in the table below. The reward functions that are being tested currently are split into sections, with the weight of each model followed by the standard deviation (std) of each parameter used in said model.

Param (Reward)	Type	Range	Description
reward/w_torque	float	[0,1]	Weight of torque model in reward function
reward/torque	list	[0,inf)	Std used in gaussian for torque
reward/w_angle	float	[0,1]	Weight of angle model in reward function
reward/angle	list	[0,inf)	Std used in gaussian for angle
reward/w_advisor	float	[0,1]	Weight of advisor model in reward function
reward/advisor_yaw_std	list	[0,inf)	Std used in gaussian for yaw
reward/advisor_vel_std	list	[0,inf)	Std used in gaussian for velocity
reward/advisor_x_std	list	[0,inf)	Std used in gaussian for X position difference
reward/advisor_y_std	list	[0,inf)	Std used in gaussian for Y position difference
reward/w_gates	float	[0,1]	Weight of gate model in reward function
reward/gate_std_x	list	[0,inf)	Std used in gaussian for X position difference
reward/gate_std_y	list	[0,inf)	Std used in gaussian for Y position difference

Once your confirguration file is as you like it, you can easily begin training RallyAI, which is described in the next section.

1.3 Training

To train RallyAI, we load the configuration parameters, build our custom gym environment and use the simulation and learning algorithm to train the policy network based off the reward function. To begin this process, navigate to `/RallyAI_v2/RL_Training/` and run

```
python train.py
```

The train logs the information to W&B based off of your project name and run name.

1.4 Testing

Testing is completed by simply loading the trained model, requesting/applying actions and updating the current state. There are no updates done in this process. To test, navigate to `RallyAI_v2/RL_Training/` and run

```
python test.py
```

CHAPTER

TWO

DOCUMENTATION

2.1 Rewards and Costs

The current reward function(s) for RallyAI are defined below. Each method is defined further below.

2.1.1 Advisor Model

When most of us learn to drive, we have an *expert* accompany us in the passenger seat. They may direct us how to control the car or correct us when we've made a mistake. This mechanism essentially makes up the advisor model.

In this method, we train using two models. One being RallyAI (who is learning to drive) while the other is an advisor (that is controlled deterministically through the user). The advisor model uses a simplified simulation, one without slip or outside forces that may act on RallyAI. This way, the expert controller can describe to our RallyAI the ideal state of the vehicle as it drives. Hopefully this leads to learning of control so that the AI can correct any extreme situation to match the *ideal case*.

For example, imagine we are driving on an icy road and as we turn the wheels begin to slip. The advisor model, which contains no ice, would not slip and RallyAI would begin to notice a discrepency in it's current state and that of the advisor. We will denote the advisor's states as *desired states* Since we want the two models to have an identical state, RallyAI would begin measures to fix this.

The reward function for the advisor model is simply the sum of the normed differences of the current states and the desired states. The *state of the states* is still very fluid and the list of observations being considered is changing day by day.

2.1.2 Additional Reward Models

The following subsections describe orther reward functions or functions used in combination with the above reward function.

Gate Model

The gate model involves running RallyAI and computing a gate we want the vehicle to pass through at time t. We use a 2 dimensional gate, eventually 3D, to compare the vehicles position in space to where we expect it. We use gaussian distribution to create a bump-shaped object in the road, where being closer to the centre of the bump returns larger rewards. The gate is updated based on the user's inputs and velocity of the vehicle in the previous timestep.

Simplified User Map

This simple reward function will be used as a precursor to the above complex reward function. It is based on the difference in user/driver input and wheel states. We also shift from punishment/cost to only positive rewards by taking the gaussian (denoted $\mathbb{N}$) of the difference. Note that $||x||_2$ is the Euclidean norm.

The twelve wheel states (4x[torque, brake and steering angle]) are used in the reward function, as well as the three user inputs (throttle, brake pedal and steering wheel).

The reward function is below.

$$R(t) = c_t \cdot torque_{diff} + c_b \cdot brake_{diff} + c_s \cdot steer_{diff}$$

where c_t, c_b and c_s are constants. Below we define the variables in the simple reward function.

$$\begin{aligned} torque_{diff} &= \mathbb{N}(||usr_{thr} - fl_{tor}||_2) + \mathbb{N}(||usr_{thr} - fr_{tor}||_2) + \mathbb{N}(||usr_{thr} - rl_{tor}||_2) + \mathbb{N}(||usr_{thr} - rr_{tor}||_2 \\ brake_{diff} &= \mathbb{N}(||usr_b - fl_b||_2) + \mathbb{N}(||usr_b - fr_b||_2) + \mathbb{N}(||usr_b - rl_b||_2) + \mathbb{N}(||usr_b - rr_b||_2) \\ steer_{diff} &= \mathbb{N}(||usr_{sa} - fl_{sa}||_2) + \mathbb{N}(||usr_{sa} - fr_{sa}||_2) + \mathbb{N}(||usr_{sa} - rl_{sa}||_2) + \mathbb{N}(||usr_{sa} - rr_{sa}||_2) \end{aligned}$$

Magnified Saltatory Reward

Defined here, the MSR algorithm magnifies *good* and *bad* rewards computed by the other function components. MSR is applied after a reward is computed, and means to accelerate positive learning. If a move results in a reward that is sufficiently better, or worse, than the previous reward, that reward is magnified to produce more good moves, and less bad moves. The algorithm is outlined in pseudo-code here.

The threshold of ν must be explored to find the value which results in the best learning.

This was attempted by Brandon earlier, but in a discrete way, whereas this method is continuous and fits better with our environment space.

Catastrophic Control Failures

Since our top priority is safety, we want to make sure the vehicle stays away from catastrophic failures, such as crashing, swerving into the other lane or leaving the road entirely. One of the ways to prevent such behaviour in an AI is by punishing the vehicle with a large negative reward.

2.2 Environment

The environment RallyAI is built within is based off of OpenAI's Gym environments. By doing this, we are able to use open source reinforcement learning packages like stable baselines (TF1.14) or stable baselines (PyTorch).

To define your own custom gym environment, you need to define five things

- Action Space
- Observation Space
- Step function
- Reset function
- Render function

The step function completed a single training step where RallyAI interacts with the environment. This includes updating observations and logging information.

The reset function resets the parameters to an initial state. In our case we have a soft reset that is used to reinitialized the vehicle, as well as a hard reset that is used between each episode.

The render function can be used to visualize the steps taken by RallyAI during training, though we do this through our simulation which is not included in the render function as of yet.

2.2.1 Action Space

The action space defines the number of outputs from the neural network. We have created a schedule of learning goals, each with a defined test case. The initial test case has an action space of dimension 1, which we apply to the vehicle's steering angle. Specifically the requested steering angle, since we are limited by the motors reaction speed.

In later test cases, we may introduce additional actions like

- Torque
- Brake
- Multiple wheel controls
- etc

2.2.2 Observation Space

The observation space defines the number of inputs to the neural network. As we attempt to optimize learning in each test case, we are adding/removing environment states to the observation space. For example, we have considered

- User inputs
- IMU data
- Desired states (advisor model)
- Distance to gate (gate model)
- etc

This is currently a very fluid aspect of RallyAI.

2.3 Setup of W&B

Follow the instructions to install Weights and Biases visualization tools here. Weights and Biases is a visualization tool for ML projects. It allows for a clean, succinct way to show results, as well as a the ability to log vast amounts of different data types.

To initialize our project with W&B, we use

```
import wandb
wandb.init(project, config_dict)
```

where project is the project title and config_dict is a dictionary containing all run configurations.

To add a log call into RallyAI, use the command

```
wandb.log(info_dict)
```

where info_dict is a dictionary containing all of the data and variables you wish to log.

2.4 Useful Links

Github - Potential Motors repositories, including RallyAI's code and documentation.

W&B - RallyAI projects and visulizations.

Miro - Potential Motors Miro boards, including documentations and discussions on RallyAI, reward functions and ROS architecture.

CHAPTER

THREE

NEWTONRC

3.1 ROS

The NewtonRC utilizes a ROS network to connect RallyAI and the phsyical vehicle. The GWagon will have a similar network, but using ROS2. The connections consist of a network of nodes. Each node can publish information over a topic (think of this as a radio station) or subscribe to said topic.

3.1.1 Install ROS

To install ROS on your machine, follow the instructions here. RallyAI requires the ROS package to be at least the `ros-melodic-desktop`. Once it is installed, we can begin setup.

3.1.2 Setup ROS

First, we need to `source /opt/ros/melodic/setup.bash`. Initialize a workspace and within, create a directory named `src`. Then, from within the workspace, initialize the workspace with `catkin_make`. This generates the files necessary for a ROS workspace.

Note: Sourcing ROS is necessary in each new terminal that is opened. If you consider this to be annoying, as I do, you can add the source command directly to the bashrc file with `echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc` (if using Ubuntu OS).

Now we can begin to build ROS packages inside the `src/` directory. Each subdirectory here is considered a package and RallyAI has two such packages: `~/ws/src/messages` and `~/ws/src/newton`. In the following sections we outline each of these packages.

Note: In order for the files inside `newton/` to access the msg files in `messages/` we must source from the workspace directly with `source ./devel/setup.bash`. This can also be added to the bachrc file with `echo "source ~/ws/devel/setup.bash" >> ~/.bashrc`.

3.1.3 Custom Messages

The package `~/ws/src/messages/` contains the messages for the `~/ws/src/newton/` package. The repo can be found here and can be simply cloned into `~/ws/src/`.. There exist standard messages built into the ROS package, like `Int`, `Int8`, `Float32`, `String`, etc. However ROS also allows for the definition of custom messages which work consistently between Nodes, using Python or C. It also gives us the ability to label the data within the message.

The messages include

Message	Description
actions	RallyAI output actions
controls	Message/actions sent to motors
determ	Determistic User actions
joystick_user	User actions from joystick
sim_states	State, input for simulation
states	State/observations, input for RallyAI
user	User actions from steering wheel/throttles

To create a new custom message, follow these steps

1. Navigate to `ws/src/messages/msg/`
2. Create a message file, `cust_msg.msg`

- Within the file list each data type and feature (i.e. `dataType featureName`) on a new line
- The 'dataType' can be anything from stdMsgs in ROS.

3. In the node file(s) that use `type.msg`, add the line `from messages.msg import type`
4. To initialize the message, write `msg = cust_msg()`
5. To assign values to features, you can write `msg.featureName = val`

- Confirm the type of `val` is the same as that of `featureName`

6. Navigate to the workspace directory `ws/` and run `catkin_build`

3.1.4 Control Network

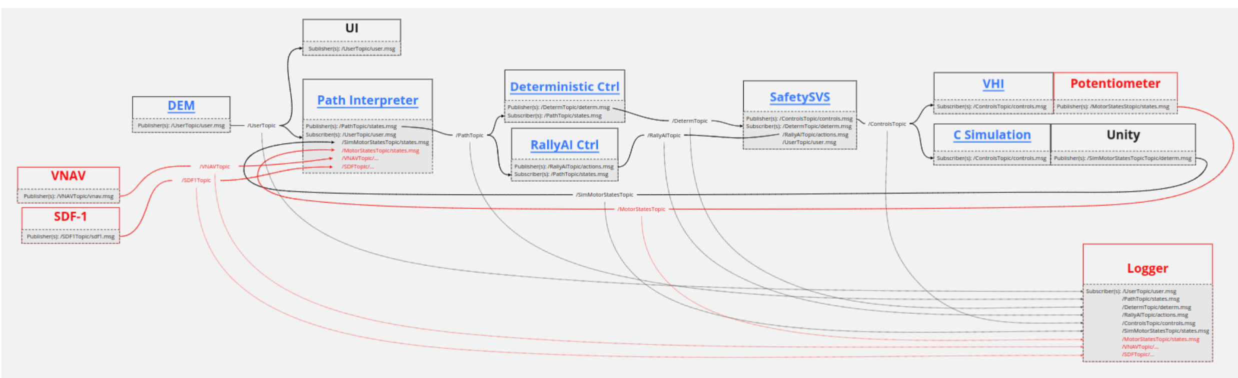
The package `~/ws/src/newton/` contains the ROS nodes which comprise the nodes in the network. For this package, you will have to compile all the nodes seperately and build the package. This is explained in the section below. The list of nodes in this package are in the table below.

Nodes	Description	File Location
Driver Experience Module	Driver experience module, contains user input modules	Potential-Motors/DEM/DEM_Node.py
Path Interpreter	Inpterprets the path based off of the user inputs and current state of the vehicle	Potential-Motors/Path-Interpreter/PathInterp.py
Deterministic	Deterministically controls the vehicle	Potential-Motors/Deterministic/DetermCode.py
RallyAI	Controls the vehicke using RallyAI	Potential-Motors/RallyAI_Node/tc_0a.py (tmp)
SafetySVS	Accepts the actions from RallyAI and determines if they are safe	Potential-Motors/SafetySVS/SafetySVS.py
Simulation	Contains the bicycle simulation for visualization, if not implemented/testing in Newton	N/A
Motor/Servo	Sends controls to motor servos	N/A

As mentioned above, these nodes communicate with one another by listening and sending information across *topics*.

For example, one such topic is called the */UserTopic* and the DEM node sends the driver's inputs across this signal. The path interpreter node listens to the same topic, and uses the information it recieves to compute the current path of the vehicle. To see the full network of topics and nodes, please checkout our miro board ROS Architecture.

Below is a diagram of the Newton's ROS network, with topics and messages included. Nodes and topics in red have not yet been implemented.



Clicking on the image above sends you to the diagram on Miro.

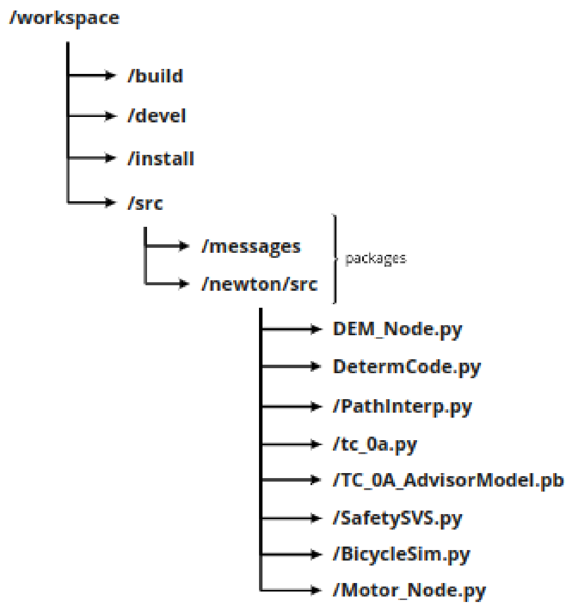
3.1.5 Building Projects

Inside the workspace dir `~/ws/src/`, we will create a package using the command below.

```
catkin_create_pkg proj_name rospy std_msgs messages
```

In our case, we chose the project name to be `newton`. The strings following the project name consist of the packages- Now navigate to `/newton/src/` and download the files from the node table above. Also download the weights for RallyAI that are also inside `Potential-Motors/RallyAI_Node/` with the same name as the RallyAI node.

After this step, the entire workspace should have the following structure.



Lastly, within `~/ws/src/newton/`, we must edit two files (`CMakeLists.txt` and `package.xml`) to allow access to the custom messages.

Inside `~/ws/src/newton/CMakteLists.txt`, there are a number of changes we need to make. Note that some of the functions need to be uncommented. Follow the list below.

1. Under the function `find_package(...)`, add `message_generation` and `messages` within the brackets
2. Uncomment the function `generate_messages` and the lines withing the brakctes
3. Inside the function `catkin_package(...)` uncomment the line `CATKIN_DEPENDS` and add `message_runtime` to the end of the line

Inside `~/ws/src/newton/package.xml`, we only need to uncomment the lines

```
<build_depend>message_generation</build_depend>
...
<exec_depend>message_runtime</exec_depend>
```

Finally, we can back out to the workspace directory and run `catkin_make`

3.1.6 Testing

To run the entire network, please follow the commands below. Notice each numbered step requires it's own terminal window.

0. Run `roscore`
1. Plug the steering wheel into your computer

a. Run `python DEM.py`

b. Follow the instructions in the terminal

c. The default driving mode is deterministic, please press the circle button to switch to AI mode

2. Run `python PathInerp.py`

3. Run `python DetermCode.py`
4. Run `python RallyAI_node.py`

a. Replace `RallyAI_node` with the test case you'd like to load in (i.e. `tc_0a.py`)

5. Run `python SafetySVS.py`
6. Run the simulation or hardware node

a. (Sim) Run `python BicycleSim.py`

b. (Real) Run `python VHI_Beaglebone.py`

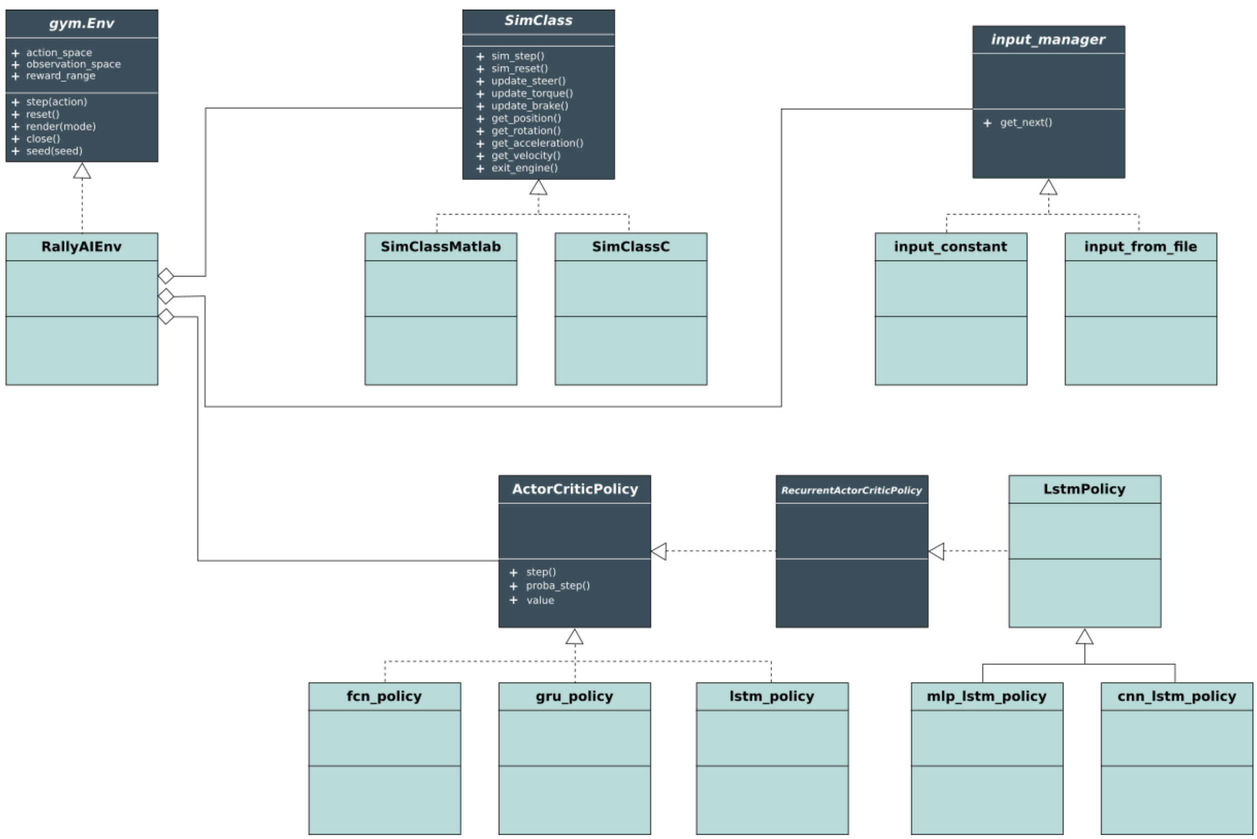
If everything is working properly, you should be able to control the simulation or Newton hardware from the steering wheel plugged into the computer.

Note: To see what information is being sent over any topic, you can open a new terminal and simply write `rostopic echo /ROSTopic` and insert any of the topics defined above.

CHAPTER

FOUR

CLASS DIAGRAM



CHAPTER

FIVE

MODULES

5.1 RallyAI.envs package

5.1.1 Class RallyAI_env

class `RallyAI.envs.RallyAI_env.`**`RallyAIEnv`**

Bases: `gym.core.Env`

RallyAI is a DNN which accepts sensor data about the area surrounding the car, and the current state of the car, and aims to follow the input instructions. This includes the correcting outside forces, resulting from extreme conditions, or advanced driving maneouvers. The key is that the AI will "Follow the User's path" and will not make it's own decisions on the direction or speed of the vehicle.

This is a custom OpenAI gym environment, which includes tailored functions for stepping through the environment, rendoring each step, calculating a reward for the RL algorithm and resetting once an episode is done.

The updated action and observation spaces, along with the reward function, can be found at the following link:

https://docs.google.com/document/d/1yjIrAkPV8bbctkYULEsw4zAM6ik8zAfjlXaI1mnUYMc

`get_reward`(*action*, *log*)

Get reward for current states and actions.

Parameters

- **`action`** (*`Pandas DataFrame`*) – Set of actions comming from the NN.
- **`log`** (*`Boolean`*) – True if you want to log to wandb.

Returns (Step reward, Reset simulation).

Return type (float, boolean)

`log_handler`(*actions*)

Logs to WandB / console. If constants.run_on_aws is true, logs will be printed directly to console. Logs will be sent to WandB otherwise.

Logs:

- user inputs,
- vehicle states,
- actions,
- expert states - if advisor reward is selected
- next gate position - if gate reward is selected

Parameters **actions** (*Pandas Dataframe*) – Actions proposed by the neural network.

render (*mode='human'*)
Renders time step and current state of environment.

reset ()
Resets environment variables and simulation.

Returns Initial state

Return type Pandas DataFrame

soft_reset ()
Resets the environment variables, but not the step count.

step (*action*)
Update states on simulation, evaluate actions, get reward. . .

Parameters **action** (*Numpy array*) – Set of actions to act on.

Returns Observations, Reward, Done, Info

Return type [Numpy array, float, boolean, dict]

update_advisor_sim ()
Update reference simulation - the one used as expert driver

update_gate ()
Updates the next gate coordinates.

update_reward_params ()
Curriculum learning - updates the standard deviation every time an episode reward value is reached. Makes the problem more challenging, by refining results.

update_states (*action*)
Update all observations (current states) using current actions and results from simulation step. :param action: Actions comming from the NN, see common.defs for further explanation. :type action: Pandas DataFrame

5.2 SimClass package

5.2.1 Class SimClass

class SimClass.SimClass.**SimClass**
Bases: `abc.ABC`

Abstract class describing the basic communication between simulation and RL agent.

Whenever a new simulator is available, we need to create a new implementation of SimClass for working with that specific simulator. Once implemented, you just need to create a new instance of that implementation inside RallyAI_env.py. No need for more adjustments.

abstract exit_engine ()
Close communication with the simulator

abstract get_state_dots ()
Get the last changes in vehicle's states: position, velocity, acceleration and rotation.

Returns Last changes in states.

Return type Pandas DataFrame

abstract get_states()
Get the vehicle's updated states: position, velocity, acceleration and rotation.

Returns Updated states.

Return type Pandas DataFrame

abstract request_actions(*actions*)
Request a new value for Torque, Brake and Steering.

Parameters actions (*Pandas DataFrame*) – Set of requested actions.

abstract sim_reset(*custom=None*)
Reset the simulator.

Parameters custom (*Pandas DataFrame*) – When set to None, reset all initial values to zero. Otherwise, custom contains initial values.

abstract sim_step()
Update the simulator, update every single variable to be accessed from it.

Implement communication to/from simulator inside this method.

5.2.2 Class SimClassMatlab

class SimClass.SimClassMatlab.**SimClassMatlab**(*prj_path*, *prj_name*, *sim_name*)
Bases: *SimClass.SimClass.SimClass*

exit_engine()
Exits Matlab engine.

get_state_dots()
Get the last changes in vehicle's states: position, velocity, acceleration and rotation.

Returns Last changes in states.

Return type Pandas DataFrame

get_states()
Get the vehicle's updated states: position, velocity, acceleration and rotation.

Returns Updated states.

Return type Pandas DataFrame

request_actions(*actions*)
Request a new value for Torque, Brake and Steering.

Parameters actions (*Pandas DataFrame*) – Set of requested actions.

sim_reset(*custom=None*)
Reset the simulator.

Parameters custom – When set to None, reset all initial values to zero. Otherwise, custom contains initial values.

sim_step()
Update the simulator, update every single variable to be accessed from it.

5.2.3 Class SimClassC

class `SimClass.SimClassC.`**`SimClassC`**

Bases: *`SimClass.SimClass.SimClass`*

Class describing the basic communication between simulator in C and RL agent.

There are two options for stablishing the communication between the simulation and this script:

1. **Shared Library**: The shared library will be saved either on /usr/lib or BasicSimulation/C. If not available, the second option will be attempted.

2. **Socket communication**: The simulation is the server and must be started before running this script. Communication will be held on port 2300.

class actionStructure

Bases: `_ctypes.Structure`

`exit_engine``()`

Close communication with the simulator

`get_state_dots``()`

Get the last changes in vehicle's states: position, velocity, acceleration and rotation.

Returns Last changes in states.

Return type Pandas DataFrame

`get_states``()`

Get the vehicle's updated states: position, velocity, acceleration and rotation.

Returns Updated states.

Return type Pandas DataFrame

`request_actions`(*actions*)

Request a new value for Torque, Brake and Steering.

Parameters **`actions`** (*`Pandas DataFrame`*) – Set of requested actions.

`sim_reset`(*custom=None*)

Reset the simulator.

Parameters **`custom`** (*`Pandas DataFrame`*) – When set to None, reset all initial values to zero. Otherwise, custom contains initial values.

`sim_step``()`

Update the simulator, update every single variable to be accessed from it.

Implement communication to/from simulator inside this method.

class statesStructure

Bases: `_ctypes.Structure`

class vehicleParamStructure

Bases: `_ctypes.Structure`

5.3 common package

5.3.1 Constants

common.constants.**advisor_accel_std = 1**

Parameters **advisor_accel_std** (*float*) – standard deviation for acceleration in advisor reward function.

common.constants.**advisor_accel_weight = 1**

Parameters

- **advisor_accel_weight** – weight to apply to advisor reward for accurately following acceleration.
- **advisor_accel_weight** – float

common.constants.**advisor_vel_std = 1**

Parameters **advisor_vel_std** (*float*) – standard deviation for velocity in advisor reward function.

common.constants.**advisor_vel_weight = 1**

Parameters

- **advisor_vel_weight** – weight to apply to advisor reward for accurately following velocity.
- **advisor_vel_weight** – float

common.constants.**advisor_x_std = 1**

Parameters **advisor_x_std** (*float*) – standard deviation for x in advisor reward function.

common.constants.**advisor_x_weight = 1**

Parameters **advisor_x_weight** (*float*) – weight to apply to advisor reward for accurately following X position.

common.constants.**advisor_xdot_std = 1**

Parameters **advisor_xdot_std** (*float*) – standard deviation for derivative in X position in advisor reward function.

common.constants.**advisor_xdot_weight = 1**

Parameters

- **advisor_xdot_weight** – weight to apply to advisor reward for accurately following X derivative.
- **advisor_xdot_weight** – float

common.constants.**advisor_y_std = 1**

Parameters **advisor_y_std** (*float*) – standard deviation for y in advisor reward function.

common.constants.**advisor_y_weight = 1**

Parameters **advisor_y_weight** (*float*) – weight to apply to advisor reward for accurately following Y position.

common.constants.**advisor_yaw_std = 1**

Parameters **advisor_yaw_std** (*float*) – standard deviation for yaw in advisor reward function.

`common.constants.`**`advisor_yaw_weight = 1`**

Parameters **advisor_yaw_weight** (*float*) – weight to apply to advisor reward for accurately following yaw.

`common.constants.`**`advisor_yawdot_std = 1`**

Parameters **advisor_yawdot_std** (*float*) – standard deviation for the derivative of yaw in advisor reward function.

`common.constants.`**`advisor_yawdot_weight = 1`**

Parameters **advisor_yawdot_weight** (*float*) – weight to apply to advisor reward for accurately following Yaw derivative.

`common.constants.`**`advisor_ydot_std = 1`**

Parameters **advisor_ydot_std** (*float*) – standard deviation for the derivative in Y position in advisor reward function.

`common.constants.`**`advisor_ydot_weight = 1`**

Parameters

- **advisor_ydot_weight** – weight to apply to advisor reward for accurately following Y derivative.
- **advisor_ydot_weight** – float

`common.constants.`**`angle_std = 0.1`**

Parameters **angle_std** (*float*) – standard deviation for angle mapping

`common.constants.`**`constant_angle = 0.0`**

Parameters **constant_angle** – For constant user input, this is the value of angle.

`common.constants.`**`constant_brake = 0.0`**

Parameters **constant_brake** – For constant user input, this is the value of brake.

:type constant_brake. float

`common.constants.`**`constant_torque = 1.0`**

Parameters **constant_torque** (*float*) – For constant user input, this is the value of torque.

`common.constants.`**`constant_velocity = False`**

Parameters **constant_velocity** (*float*) – Apply PID for constant velocity in simulation - ignore torque action.

`common.constants.`**`episode_length = 2000`**

Parameters **episode_length** – Number of timesteps on a single episode. If input_source is

set to 'file', this value will be replaced for the number of samples in the file. :type episode_length: integer

`common.constants.`**`file_source = '../../RallyAI_datasets/Raw_User_Input/Test_Cases/test_case_`**

Parameters **file_source** (*string*) – File name of the input_source, if input_source is set to 'file'.

`common.constants.`**`first_gate = 1.0`**

Parameters **first_gate** (*float*) – Sets the location for the first gate (X direction for now) `common.constants.`**`gate_dist = 0.1`**

Parameters **`gate_dist`** (*`float`*) – The distance between each gate

`common.constants.`**`gate_length = 1.0`**

Parameters **`gate_width`** (*`float`*) – Sets the length of the checkpoints (gate_length on either end of the centre)

`common.constants.`**`gate_std_x = 1`**

Parameters **`gate_std_x`** (*`float`*) – standard deviation for gate function.

`common.constants.`**`gate_std_y = 1`**

Parameters **`gate_std_y`** (*`float`*) – standard deviation for gate function.

`common.constants.`**`gate_width = 1.0`**

Parameters **`gate_width`** (*`float`*) – Sets the width of the checkpoints (gate_width on either side of the centre)

`common.constants.`**`initial_velocity = 10`**

Parameters **`initial_velocity`** (*`float`*) – Simulation initial velocity.

`common.constants.`**`input_source = 'file'`**

Parameters **`input_source`** (*`string`*) – Source from where the user input will be obtained, 'file' or 'constant'.

`common.constants.`**`log = False`**

Parameters **`log`** (*`boolean`*) – True for logging to WandB, False otherwise.

`common.constants.`**`log_frequency = 100`**

Parameters **`log_frequency`** (*`integer`*) – Number of time steps between logs.

`common.constants.`**`lstm = None`**

Parameters **`lstm`** (*`int`*) – Size of the lstm layer, if it exists.

`common.constants.`**`max_accel = 100`**

G-Wagon Simulation Parameters, from vehicle parameter excel sheet. :param maxAccel: Maximum possible acceleration of G Wagon body :type maxAccel: int

`common.constants.`**`max_vel = 100`**

Parameters **`maxVel`** (*`int`*) – Maximum possible velocity of G Wagon body

`common.constants.`**`max_wheel_angle = 35`**

Parameters **`max_wheel_angle`** (*`float`*) – Maximum steering angle in moving wheels, in degrees.

`common.constants.`**`max_wheel_change = 0.5`**

Parameters

- **`max_wheel_change`** – Maximum change in steering angle in movin wheels, in degrees.
- **`max_wheel_change`** – float

`common.constants.`**`model_path = None`**

Parameters **`model_path`** – Set the name and path to the model to be loaded for testing or continue

training. :type model_path: string

common.constants.**network_structure = [64, 64]**

Parameters **network_structure** (*List*) – Size of the layer or layers of the policy network. The value network will have the same sizes too.

common.constants.**param_sheet_loc = '../../BasicSimulation/Development_spec_sheet.xlsx'**

Parameters **param_sheet_loc** – Excel sheet with initial parameters for the vehicle, like

maximum velocity and acceleration. :type param_sheet_loc: string

common.constants.**policy_kwargs = None**

Parameters **policy_kwargs** (*Dict*) – Arguments passed into the policies used from stable baselines

common.constants.**queue_length = (1000,)**

Parameters **queue_length** (*int*) – The number of gates in the queue, will stay constant because we append new gates as we travel

common.constants.**randomize = False**

Parameters **randomize** (*boolean*) – True for training with domain randomization.

common.constants.**run_on_aws = False**

Parameters **run_on_aws** (*boolean*) – True for running on AWS (no WandB logs, print instead)

common.constants.**simulator_type = 'c'**
Type of simulation we will use for this training / testing session. :param simulator_type: 'c' or 'matlab' :type simulator_type: string

common.constants.**time_code = False**

Parameters **time_code** (*boolean*) – Determines if we want to log the run time of each code component.

common.constants.**torque_std = 0.1**

Parameters **torque_std** (*float*) – standard deviation for torque mapping

common.constants.**vehicle = 'Newton'**

Parameters **vehicle** – Determines which vehicle (Gwagon or Newton) parameters

will be used for training :type vehicle: string

common.constants.**w_advisor = 0**

Parameters **w_advisor** (*float*) – weight to apply to reward for accurate mimic of the advisor.

common.constants.**w_angle = 0**

Parameters **w_angle** (*float*) – Weight to apply to angle mapping for reward.

common.constants.**w_gates = 0**

Parameters **w_gates** (*float*) – Weight to apply to gate reward.

common.constants.**w_torque = 0**

Parameters **w_torque** (*float*) – Weight to apply to torque mapping for reward.

common.constants.**wandb_proj_name = 'generic'**

Parameters **wandb_proj_name** (*string*) – Weights & Biases project name. Default is **generic**.

`common.constants.wandb_run_name = ''`

Parameters **wandb_run_name** (*string*) – Weights & Biases run name. Empty string for a generic name.

`common.constants.whl_offset_x_front = 0.834`

Parameters **whl_offset_x_front** (*float*) – Distance from center of gravity to front tires.

`common.constants.whl_offset_x_rear = 0.834`

Parameters **whl_offset_x_rear** (*float*) – Distance from center of gravity to rear tires.

`common.constants.whl_offset_y_front = 1.138`

Parameters **whl_offset_y_front** (*float*) – Distance from center of gravity to front tires.

`common.constants.whl_offset_y_rear = 1.138`

Parameters **whl_offset_y_rear** (*float*) – Distance from center of gravity to rear tires.

Class input_manager

class `common.input_manager.`**input_manager**

Bases: `abc.ABC`

Get input values for throttle, brake and steering wheel angle.

abstract get_next()

Return next observation [User_throttle, User_brake, User_angle].

Normalize values in range [-1, 1]:

- User_throttle: -1 –> No throttle, 1 –> Full throttle
- User_brake: -1 –> No brake, 1 –> Full brake
- User_angle: -1 –> max left turn, 1 –> max right turn

Returns Dictionary with the current observation.

Return type Dictionary.

5.3.2 Class input_constant

class `common.input_manager.`**input_constant**(*throttle=1*, *brake=-1*, *angle=0*, *max_factor=None*)

Bases: *common.input_manager.input_manager*

Constant user input class.

Use this class when you need to train on input values for the full episode.

Parameters

- **throttle** (*float*) – Constant value for throttle in range [-1,1] for [backward, forward]
- **brake** (*float*) – Constant value for brake in range [-1,1] for [no brake, brake]
- **angle** (*float*) – Constant value for angle in range [-1,1] for [max angle to left, max angle to right]

- **max_factor** (*float*) – When None, the values of throttle, brake and angle will be the only output, When max_factor != 0, the value of throttle, brake and angle will be random in range [-throttle,throttle], [-brake,brake] and [-angle,angle]

get_next()

Return next observation [User_throttle, User_brake, User_angle].

Normalize values in range [-1, 1]:

- User_throttle: -1 –> No throttle, 1 –> Full throttle
- User_brake: -1 –> No brake, 1 –> Full brake
- User_angle: -1 –> max left turn, 1 –> max right turn

Returns Dictionary with the current observation.

Return type Dictionary.

5.3.3 Class input_from_file

class common.input_manager.**input_from_file**(*file_name*, *shuffle=False*, *max_values=None*, *max_factor=None*)

Bases: *common.input_manager.input_manager*

Input CSV file from where we will read input data.

The file must have the following columns:

- Throttle
- Steering

Parameters

- **file_name** (*string*) – Name of the CSV file with input values.
- **shuffle** (*boolean.*) – True to shuffle the samples read from file.
- **max_values** – Max value of throttle, brake and angle, in that specific order.
- **max_values** – [float]

get_next()

Return next observation [User_throttle, User_brake, User_angle].

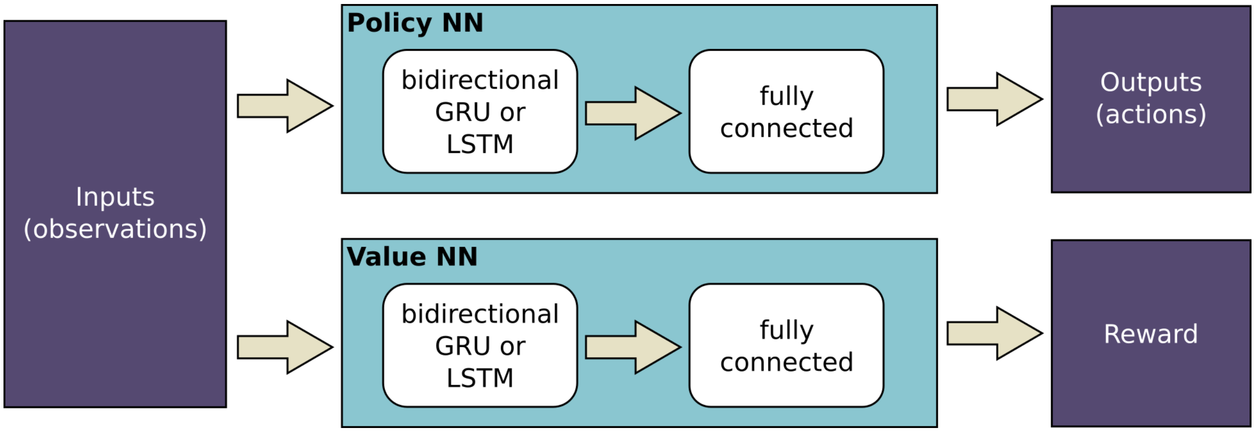
Normalize values in range [-1, 1]:

- User_throttle: -1 –> No throttle, 1 –> Full throttle
- User_angle: -1 –> max left turn, 1 –> max right turn

Returns Dictionary with the current observation.

Return type Dictionary.

Custom policies



5.3.4 Class RallyAI

class `common.policies.`**`RallyAIPolicy`**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, ***kwargs*)

Bases: `stable_baselines.common.policies.ActorCriticPolicy`

RallyAI Generic Policy. This policy cannot be instanced, it must be implemented to build the model inside the constructor.

`proba_step`(*obs*, *state=None*, *mask=None*)

Returns the action probability for a single step

Parameters

- **`obs`** – ([float] or [int]) The current observation of the environment
- **`state`** – ([float]) The last states (used in recurrent policies)
- **`mask`** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) the action probability

`step`(*obs*, *state=None*, *mask=None*, *deterministic=False*)

Returns the policy for a single step

Parameters

- **`obs`** – ([float] or [int]) The current observation of the environment
- **`state`** – ([float]) The last states (used in recurrent policies)
- **`mask`** – ([float]) The last masks (used in recurrent policies)
- **`deterministic`** – (bool) Whether or not to return deterministic actions.

Returns ([float], [float], [float], [float]) actions, values, states, neglogp

`value`(*obs*, *state=None*, *mask=None*)

Returns the value for a single step

Parameters

- **`obs`** – ([float] or [int]) The current observation of the environment
- **`state`** – ([float]) The last states (used in recurrent policies)

- **mask** – ([float]) The last masks (used in recurrent policies)

Returns ([float]) The associated value of the action

5.3.5 Class fcn_policy

class common.policies.**fcn_policy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, ***kwargs*)

Bases: `stable_baselines.common.policies.FeedForwardPolicy`

RallyAI Policy implementation for a model with fully connected layers only.

You can change number of hidden layers and their sizes in the call to model_fcn function, parameter called 'hidden_layers'.

5.3.6 Class mlp_lstm_policy

class common.policies.**mlp_lstm_policy**(*sess*, *ob_space*, *ac_space*, *n_env*, *n_steps*, *n_batch*, *reuse=False*, ***_kwargs*)

Bases: `stable_baselines.common.policies.LstmPolicy`

Policy object that implements actor critic, using LSTMs with an MLP feature extraction

You can modify the sizes of the hidden networks by modifying the numbers inside net_arch and n_lstm parameters. This policy design MUST have a shared layer between policy and value networks.

Parameters

- **sess** – (TensorFlow session) The current TensorFlow session
- **ob_space** – (Gym Space) The observation space of the environment
- **ac_space** – (Gym Space) The action space of the environment
- **n_env** – (int) The number of environments to run
- **n_steps** – (int) The number of steps to run for each environment
- **n_batch** – (int) The number of batch to run (n_envs * n_steps)
- **reuse** – (bool) If the policy is reusable or not
- **kwargs** – (dict) Extra keyword arguments for the MLP feature extraction

5.3.7 Class cnn_lstm_policy

common.train module

common.utils module

Class visualizer

PYTHON MODULE INDEX

c

r

s

INDEX

V

W