minds.ai
Accelerated Deep Learning

Enhancing HEV fuel efficiency with DeepSim

This document details a case study using the minds.ai DeepSim platform to train an RL-agent that can control the energy management system of a Hybrid Electric Vehicle (HEV). The agent is trained to maximise the fuel efficiency of an HEV over previously unseen drive cycles.

The topics covered in this white paper are:

- Reinforcement learning and hybrid electric vehicles overview.
- Adapting a simulator for use with DeepSim.
- Using DeepSim to train with the simulator.
- Training experiments.
- Summary of results.

Introduction

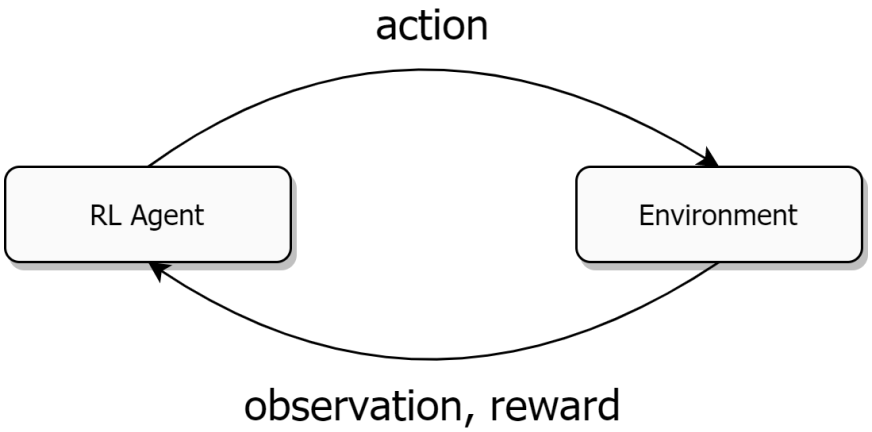
Reinforcement learning

In reinforcement learning, an agent is trained that takes actions in an environment for a particular control task. As large amounts of experience are needed to train an agent, simulations are used to model the environment and gather this experience.

During training, the simulation is evolved for a given number of simulation time steps n_s, each of size dt_s, before an action is taken, where $n_s = dt_a / dt_s$, with dt_a the time step for actions. The action a_t taken by the agent at time t depends on the current state s_t of the simulation (made up of observations about the simulation at time t), and will transition the simulation to a new state s_{t+1}. The transition of the simulation from s_t to s_{t+1} will result in a reward r_t, which indicates how 'good' the action is for some defined task. The sets of state, action, reward (s_t, a_t, r_t) values at each time step are used to train the agent, which follows a policy approximated by a neural network.

The shape of the reward is an important aspect of reinforcement learning. The reward function takes the new state of the simulation and calculates a single value that scores how 'good' the action was at achieving the desired outcome. The higher the reward, the better that that action was. For instance, for maximising the fuel efficiency of an HEV, an action that ultimately leads to

an improvement in efficiency should have a higher value than an action that leads to a reduction in fuel efficiency.



Schematic of an RL-agent interacting with the simulation environment. Actions are taken by an agent based on an observation, which results in a new observation and a reward returned by the environment.

Hybrid Electric Vehicles

A Hybrid Electric Vehicle (HEV) uses both an internal combustion engine (ICE) and an electric motor (EM) to propel the car, with an energy management system (EMS) controlling the amount of power that both the ICE and the EM provide. The EM is powered by the electric storage system (ESS), typically a battery. The current state of charge (SOC) of the battery is important: the battery depletes in order to power the motor but is recharged when the car brakes (regenerative braking), converting the kinetic energy of the car and storing it in the battery. Note that the battery is never charged externally, as would be the case for plug-in Hybrid Electric Vehicles (PHEVs). As such, the battery is never allowed to fully deplete, with the SOC maintained within a defined range.

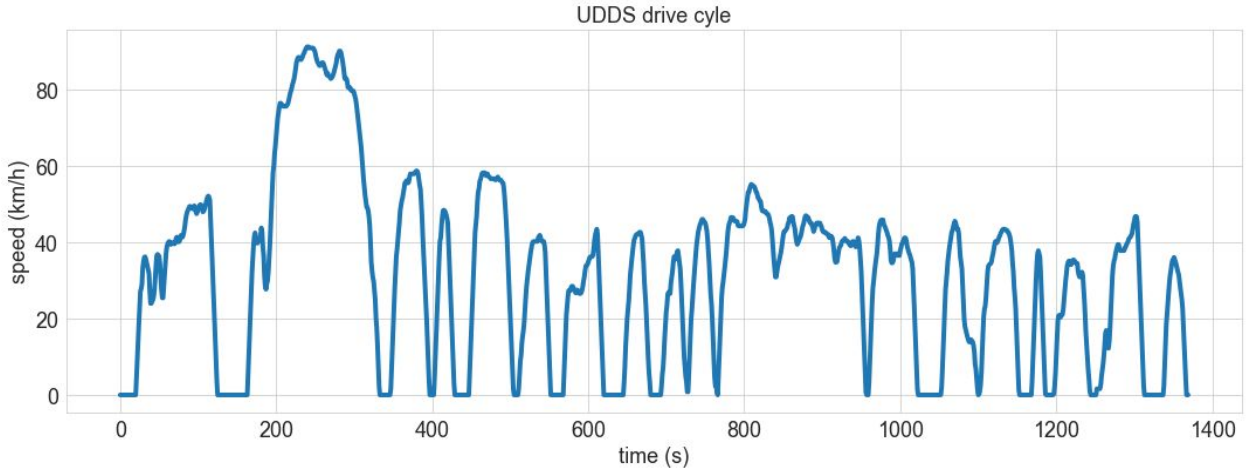
The success of an EMS can be measured by the vehicle efficiency, typically measured over a drive cycle (a scenario including the speed and incline of a car against time). There are several ways of measuring efficiency:

- **price**: how much fuel was used during the drive cycle.
- **emissions**: how much eCO2 (equivalent CO2, a measure that takes into account all warming gasses) was released during the drive cycle.
- **battery health**: while fuel usage might be minimized, this should not be at the expense of the health/lifetime of the battery which powers the EM.

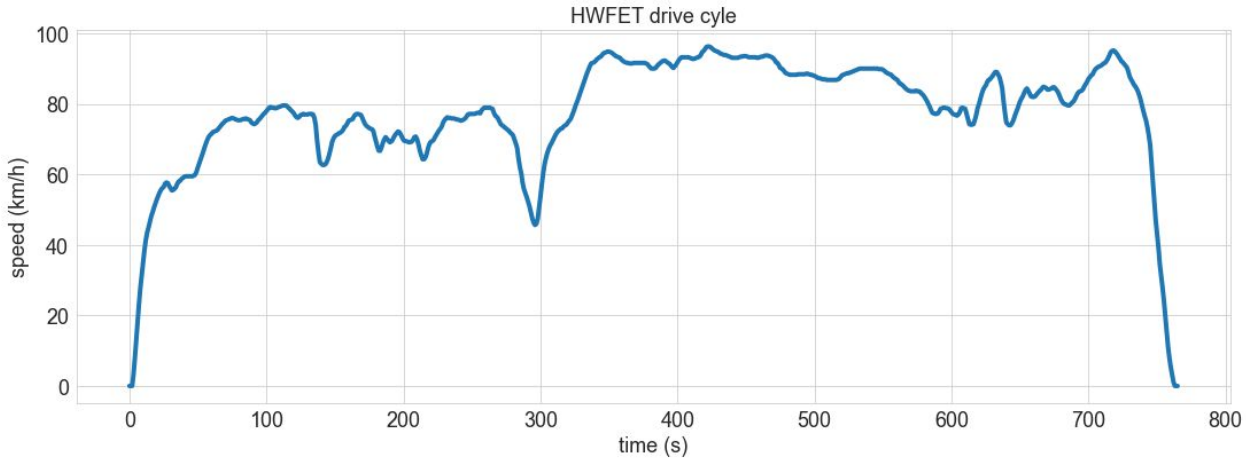
The price and emissions are usually closely related, but maybe impacted by maintaining battery health in the short term. This case study focuses on fuel consumption.

Drive cycles

Drive cycles are used to model the typical conditions a car might encounter while driving through various scenarios. For example, a standard test to measure the fuel efficiency in an urban environment is the so-called Urban Dynamometer Driving Schedule (UDDS) drive cycle, where the car drives on a flat road (zero incline/descent) with a speed profile as shown below. The car accelerates and brakes frequently, mimicking the conditions found in a city with frequent traffic lights etc.



To simulate highway driving conditions, a commonly used scenario is the Highway Fuel Economy Test (HWFET), shown below. This also simulates a scenario with zero incline/descent, with much less braking/acceleration than the UDDS scenario, and higher speeds throughout.



The RL-agent is trained on many different drive cycles, with the UDDS, HWFET and the US06 (simulating aggressive braking/acceleration) drive cycles used as unseen scenarios for testing a trained agent.

Adapting a simulator

This case study uses FASTSim, which simulates the drive cycles of HEVs in order to measure their fuel efficiency. FASTSim is a Python-based simulator and was developed by the National Renewable Energy Laboratory. Details can be found here.

In a simulation, an HEV follows a predetermined drive cycle, with the fuel efficiency, in units of miles per gallon (MPG), measured at the end of a simulation episode (that is, a completed drive cycle). At each timestep, the EMS sets the power output from both ICE and EM to meet the power required to drive at the desired speed. The RL-agent trained here will replace the built-in EMS, setting the amount of power that the ICE should provide (as a fraction of the maximum power it can provide), with the EM making up the remainder. As such, the actions that the agent will take are in the range $0 \rightarrow 1$, where 0 means no power from the ICE, and 1 means all available power from the ICE is used.

In order to adapt a simulator for use with reinforcement learning, it is necessary to repeatedly 'pause' the simulation, so that an agent can take an action to control a given simulator parameter (or to take actions, when the agent controls multiple parameters), which here will be the fraction of power from the ICE. DeepSim is compatible with OpenAI Gym environments to enable interactions with a simulator, an outline of which is given below. Full details can be found in the appendix.

Gym Environment

The OpenAI Gym environment has the following functions that are used when training an RL-agent:

- **init**: initializes the simulator environment. This involves setting parameters that will not change over different simulation runs e.g. sets the path to read data from. The action and observation spaces are defined during the initialization.
- **reset**: resets a simulator environment back to the beginning e.g. read in a drive cycle, set time to t=0, initialize any variables.
- **step**: step a simulator environment forward one action time step along a drive cycle. The RL-agent provides an action at *time* = t, which evolves the simulation to *time* = $t+1$. The observation and reward due to the action are returned to the agent.

The FASTSim simulator was adapted to run as a Gym environment, which then easily allows for RL-training via DeepSim.

Reward function

The final ingredient required to train the agent is the reward function. The aim of this case study is for the RL-agent to control the EMS of the car to maximise fuel efficiency, measured in miles per gallon. This is given by:

```
miles per gallon = distance travelled in miles / fuel consumed in gallons
```

Assuming that the RL-agent does not cause the HEV to fail to meet the desired driving speed, and safeguards can be built in to prevent this, the numerator above should be constant for a given drive cycle and therefore only the fuel consumption should change. That is, in order to maximize fuel efficiency, fuel consumption must be minimized. Two options present themselves:

i) set the reward to be the inverse fuel consumption at each timestep, or
ii) use the negative fuel consumption as the reward.

For both of these, maximising the reward will minimize consumption. The latter is preferable to the former, as the inverse fuel consumption is undefined for timesteps where no fuel is consumed (e.g. the car is not moving) and the cumulative reward is not well correlated with the aim, that is, the fuel efficiency over an entire drive cycle. The reward can then be defined as:

```
reward = - instantaneous fuel consumption
```

Note that the reward is always a penalty (always less than 0), with higher consumptions resulting in larger penalties.

In addition, in order to speed up training, a constant value is subtracted from the reward at each timestep. The fuel consumption using the FASTSim built-in EMS is used for this constant, such that the reward becomes the improvement in fuel consumption compared to the baseline value:

```
reward = - (instantaneous fuel consumption - baseline instantaneous fuel consumption)
```

Note: Actually, the effective fuel efficiency is used that takes into account both fuel and battery consumption by converting them into the same units of miles per gallon.

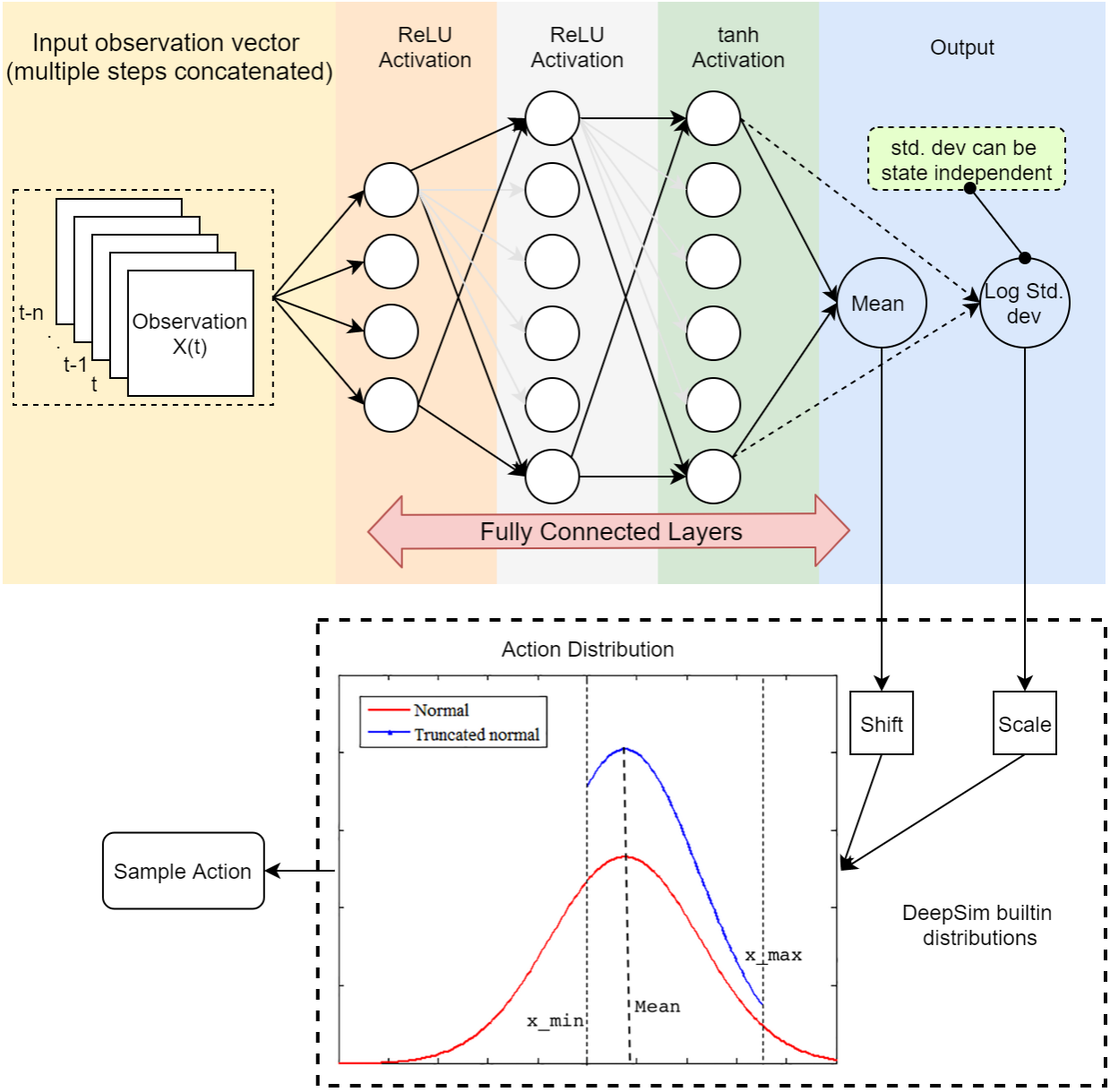
Training with DeepSim

Training Algorithm

For training the neural network we use the Proximal Policy Optimization (PPO) algorithm. This algorithm is included within the DeepSim framework and has been applied to a wide range of use cases and is one of the currently most used RL training algorithms.

Network for action prediction

The schematic below shows the fully connected neural network and the distribution used for taking actions. Details of input and layer sizes used for the experiments below can be found in the appendix.



Training configuration

When training an RL-agent with DeepSim, the user provides configurations for both DeepSim itself and the simulator that is being used. The DeepSim configuration contains parameters that define the neural network that approximates the policy that the agent follows, the number of training steps to take, the number of simulations to run for each training step, etc. The simulation configuration for FASTSim contains parameters such as which drive cycles to train

on, which vehicle to use, what parameters to use as the observation of the simulation state, etc.. See the appendix for the full training configuration used for this case study.

Note that in this case study we do not carry out hyperparameter tuning, using the same parameters in each experiment. As such, we only use train and test drive cycles, rather than also using validation scenarios to tune hyperparameters.

Inference

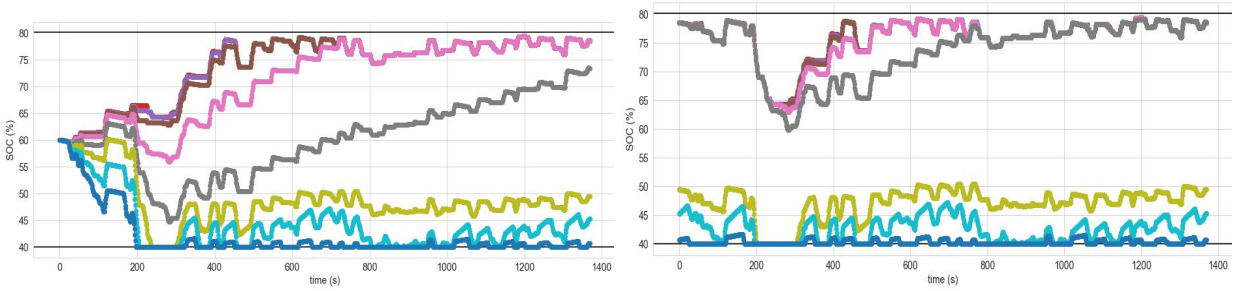
The inference is carried out using the trained agent on previously unseen drive cycles. While during training actions are picked at random from the policy distribution, during inference the most likely action is used for a given observation. For the normal distribution, this is the mean value, while for the truncated normal distribution it is the mean of the underlying normal distribution.

Charge Balancing

To avoid a biased fuel efficiency because of agents that might have learned to simply deplete the battery over a fixed drive cycle, we measure fuel efficiencies using charge balancing. Charge balancing means that a steady state solution for the state of charge (beginning state of charge = end state of charge) is found for the particular drive cycle and trained agent. Using this approach, the measured fuel efficiency is sustainable and is not due to the agent simply causing the battery to discharge over the course of a drive cycle.

This steady state can be found by running multiple consecutive simulations on a single drive cycle using a trained agent, in which the initial state of charge is kept from the previous simulation until there is no net discharge over a single simulation of the drive cycle. In the determined steady state, the initial and final state of charge of the battery is balanced. That is, there is no net charge or discharge over the course of a drive cycle. Using this approach, the measured MPG is sustainable and is not due to the agent simply causing the battery to discharge over the course of a drive cycle.

Note that when using charge balancing, the effective fuel efficiency is the same as regular fuel efficiency, as there is no net charge or discharge over the course of a drive cycle.



State of charge vs time for various fixed actions (colors) in a single drive cycle. Here it is shown for fixed actions for illustration purposes, but usually charge balancing is determined for a particular trained agent and drive cycle.
Left: Not charge balanced. The state of charge can be unsustainable, as an action results in severe depletion of the battery, such that further driving will be adversely affected.
Right: Charge balanced. The state of charge is sustainable, as for the particular constant action (color) and drive cycle, the battery has no net discharge.

Training dataset generation

The RL-agent is trained on numerous drive cycles, with the final performance measured using previously unseen test drive cycles (UDDS, HWFET, and US06). The training scenarios should simulate realistic driving conditions (e.g. urban or highway driving) while being sufficiently different from the test scenarios. To construct training scenarios, we split each of the UDDS, HWFET and US06 scenarios into separate parts (taken as a region where the car accelerates from a stand-still and then comes to a stop), permute the parts (i.e. shuffle the order) and scale the speed in each part at random (by values between the maximum and minimum speeds in the input drive cycle). The resulting drive cycles contain speed profiles that are unlike the input drive cycles used to make them.

Training experiments

In this section, details of various training experiments are given.

Baseline EMS

In order to quantify the performance of an RL-agent, a suitable baseline metric must be defined. For the case study considered here, an obvious candidate is the performance of the built-in EMS that FASTSim uses as default. Details can be found in the appendix.

Experiment 1

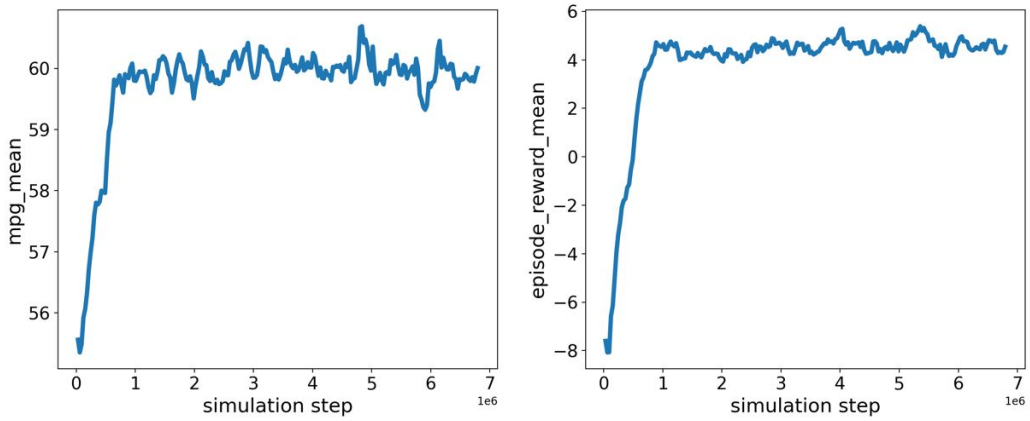
Training details: The agent is trained on 100 variants of the UDDS drive cycle, generated using the algorithm explained in the previous section. In order to mimic a long drive cycle, each training episode is started from the same state of charge at which the previous episode had ended. Full configuration details can be found in the appendix.

Evaluation: The agent is evaluated on the original UDDS cycle with charge-balancing. Charge balance is necessary here as it tries to emulate the real-world scenario where the agent drives with the same driving conditions daily and hence the battery reaches a stable state-of-charge in the long-term.

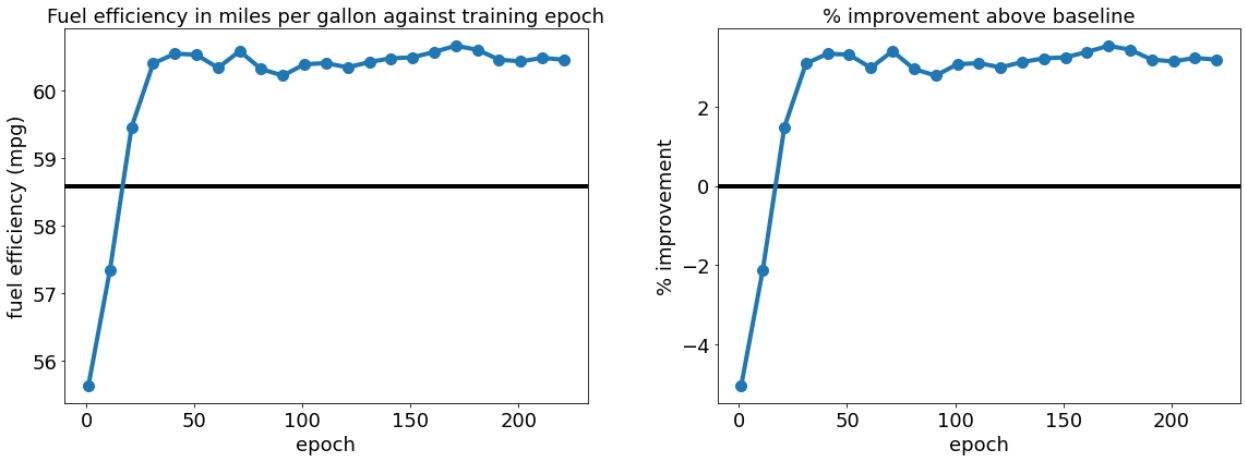
Evaluation Metrics: The trained agent is evaluated using effective fuel efficiency that takes into account both fuel and battery consumption by converting them into the same units of miles per gallon. Note that when using charge balancing, this is the same as regular fuel efficiency, as there is no net charge or discharge.

Results:

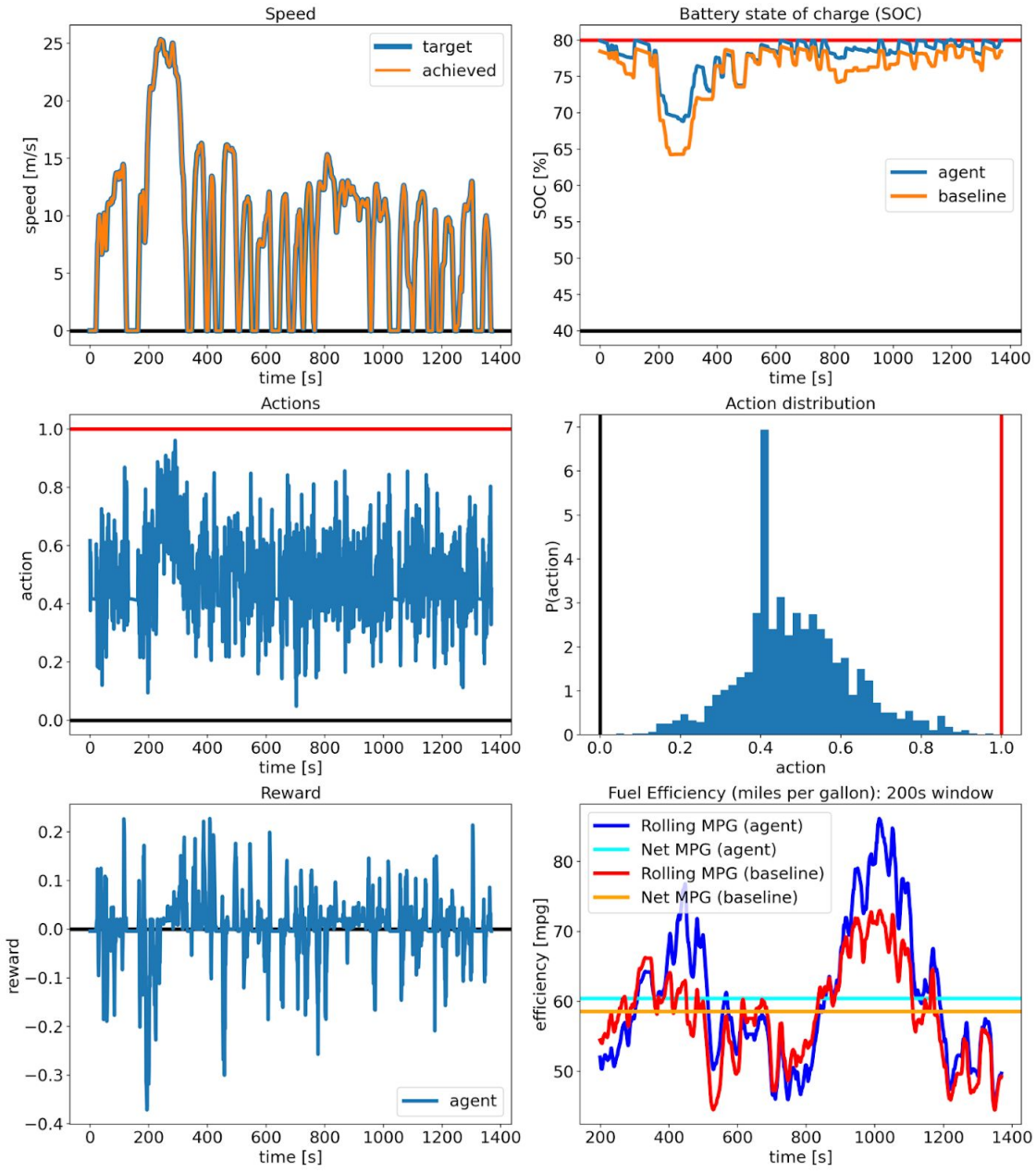
- The figure below shows the mean effective fuel efficiency and the reward function respective to the number of simulation steps taken during training. As can be seen, they both rise gradually with each step until they saturate.



- The figure below shows the performance of the trained agent on the UDDS drive cycle through various epochs of training. Agent performance can be seen to rise and saturate to a fuel efficiency which is **~ 3.2% higher** than the value achieved by the baseline EMS. Note also, this cycle was not used during training runs, hence this is an **unseen scenario** so far as the agent is concerned.



- The figure below shows the performance of the trained agent for Epoch 221 over the UDDS drive cycle against time. The different subplots show (from top left to bottom right):
    - The UDDS drive cycle target speed and the speed achieved by the agent. The simulator ensures that the agent is always able to achieve the desired speed.
    - Battery state of charge (SOC) in the HEV for both the trained agent and the baseline EMS. SOC gets balanced over time, hence it stays unchanged between the beginning and end of the drive cycle.
    - Actions taken by agent over the drive time. High actions imply increased usage of engine power (hence increased fuel consumption) to meet the desired speed.
    - Histogram of actions taken over the drive cycle (for the inference results shown here, this is the mean of the underlying normal distribution). The peak is observed near ~0.4 of max engine fuel consumption.
    - Reward at each time step in the drive cycle.
    - Rolling average of fuel efficiency over 200s window for both baseline EMS and trained agent. Also shown is the overall fuel efficiency for the two (horizontal lines). This shows the improvement in performance due to RL.



Experiment 2

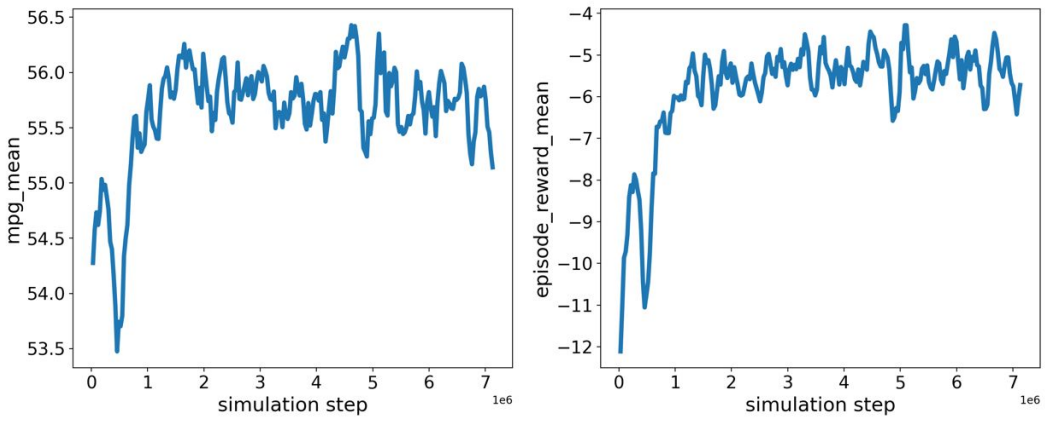
Training details: The agent is trained on the same 100 variants of UDDS drive cycle as experiment 1. However, instead of emulating one long drive cycle, each training episode starts from a random state-of-charge in the permitted range. This results in the agent seeing the whole range of SOC during training which should help in better adaptation to new scenarios.

Evaluation: The agent is evaluated on the unseen original UDDS drive cycle with multiple initial SOCs. Charge balancing is skipped to observe agent actions in different SOC regions. This evaluation is in line with the training and is used to assess the robustness of the agent.

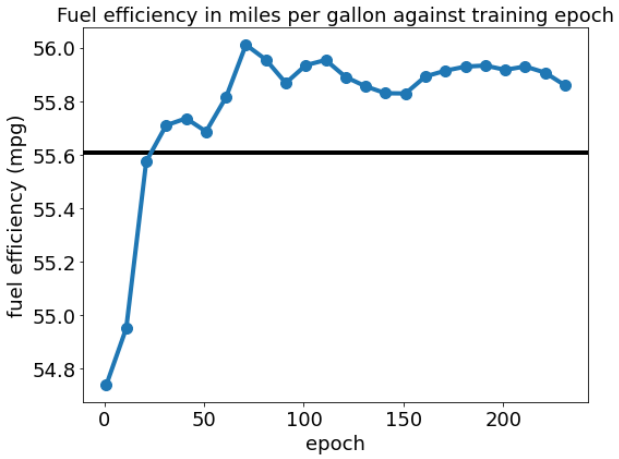
Evaluation Metrics: The trained agent is evaluated using the mean of effective fuel efficiencies for initial SOCs in the range of 0.4 to 0.8 (both included) with intervals of 0.05.

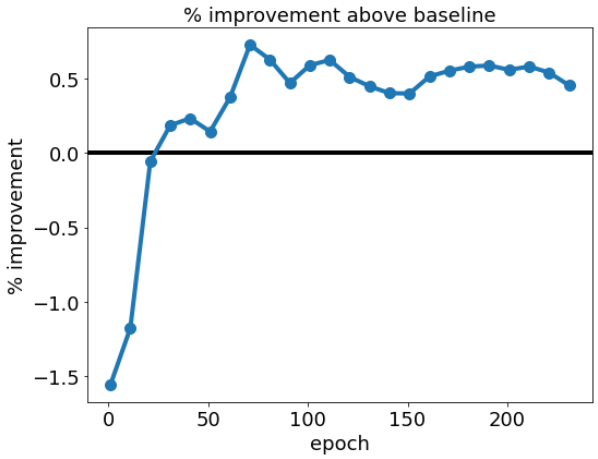
Results:

- The figure below shows plots of the mean effective fuel efficiency and the reward function respectively against the number of simulation steps. The relative reward rises and saturates to about -5. This is to be expected, as the negative rewards indicate that performance is worse than the baseline reward calculated using charge balancing. As no balancing is used here, there are scenarios where the SOC will increase over the simulation, giving a lower MPG.



- The plot below shows the evaluation of the trained agent over the unseen UDDS cycle for different epochs using the evaluation metric of average MPGGE explained above. Agent performance improves and saturates to nearly **0.5% increase** over the baseline EMS.





Experiment 3

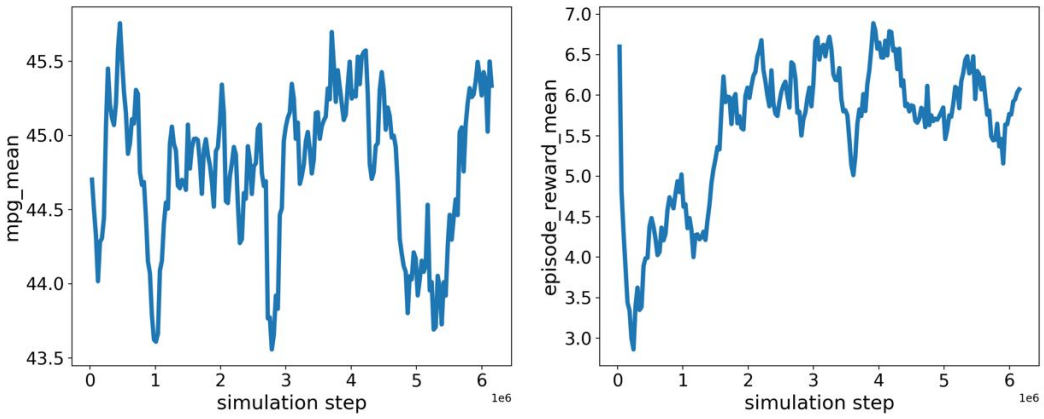
Training details: The agent is trained on 150 variants of UDDS, US06 and HWFET drive cycles mixed together and generated using the same algorithm as used previously. In order to mimic a long drive cycle, each episode starts from the same state of charge on which the previous episode had ended. Full configuration details can be found in the appendix.

Evaluation: The agent is evaluated on the 3 original cycles, using charge-balancing. Since the training takes into account variants of all these cycles, it follows that the agent is tested on all 3, to check for performance improvement in each cycle individually. Again charge balancing is necessary in order to measure sustainable fuel efficiencies.

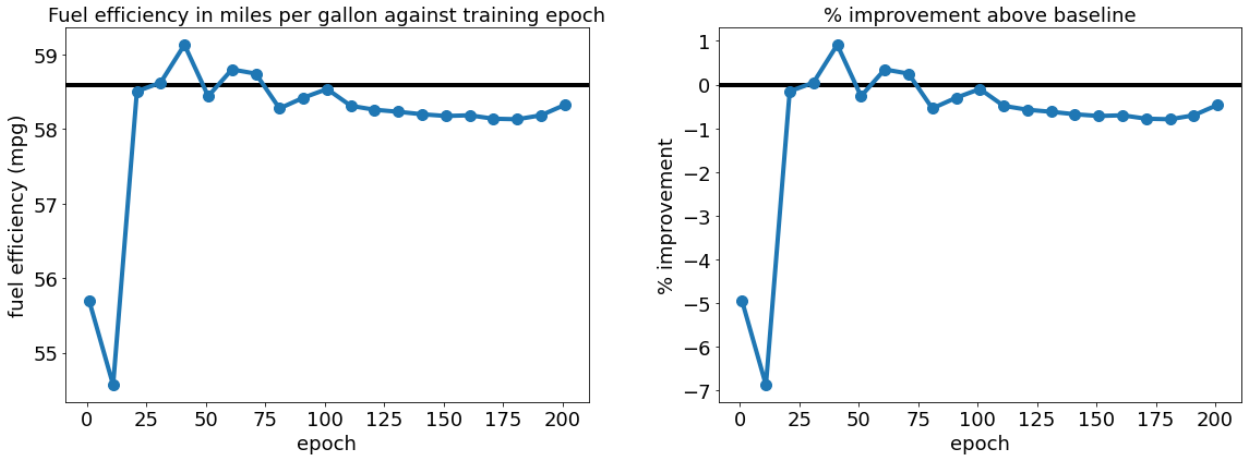
Evaluation Metrics: The trained agent is evaluated using effective fuel efficiency in all 3 scenarios.

Results:

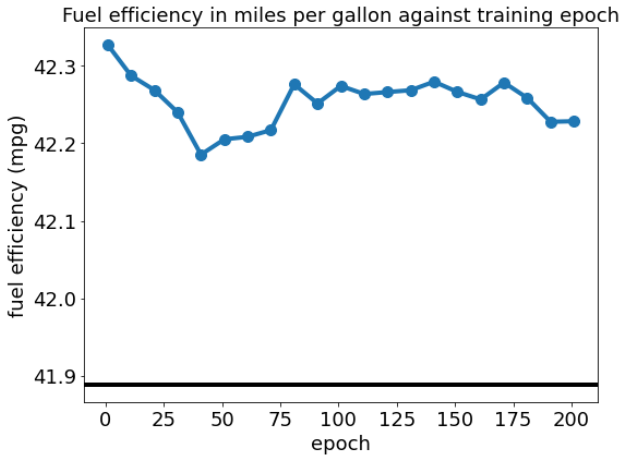
- The figure below shows the mean effective fuel efficiency and reward function respective to the training step. Due to the mixed nature of the training scenarios, the reward increase is slightly irregular.

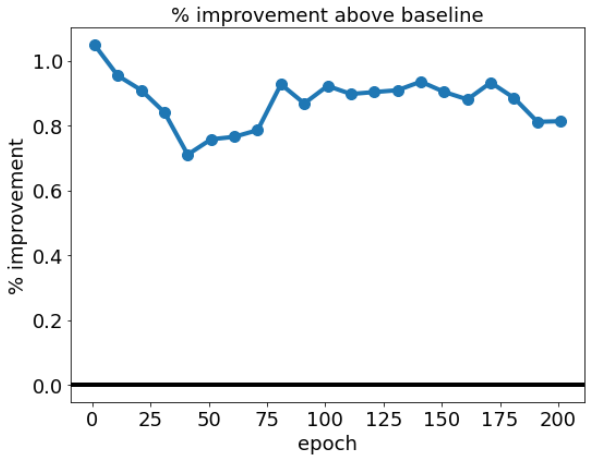


- The figure below shows the performance of the trained agent on the unseen UDDS drive cycle through various epochs of training. The drop in performance over the steps, compared to the baseline value can be explained by a similar amount of performance increase for the other cycles. Hence while the agent is slightly worse off for the UDDS cycle, it is better adapted to drive in different scenarios rather than just one (i.e. it can adapt to a highway or urban driving).

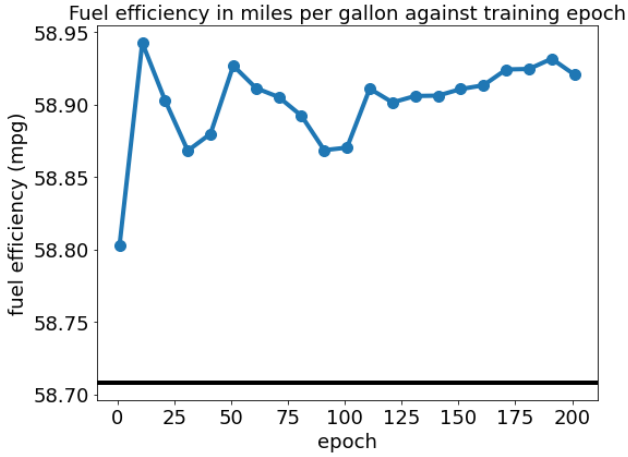


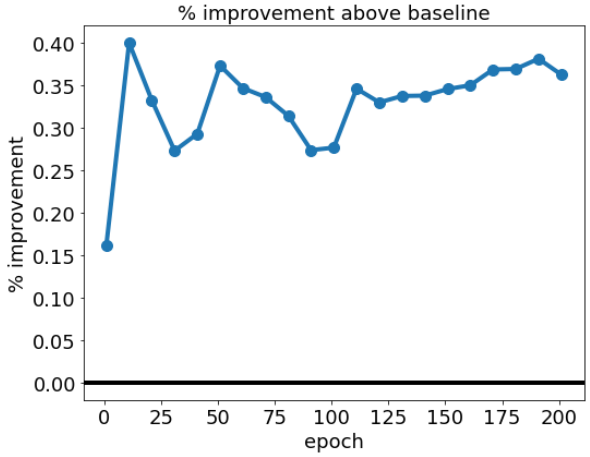
- The figure below shows the performance of the trained agent on the unseen US06 drive cycle through various epochs of training. The achieved performance of the trained agent is **0.8% higher** than the baseline EMS.





- The figure below shows the performance of the trained agent on the unseen HWFET drive cycle through various epochs of training. The achieved performance of the trained agent is **0.4% higher** than the baseline EMS.





Summary

One can draw the following conclusions from the experiments above:

- An RL agent trained for urban driving outperforms the baseline EMS in terms of effective fuel consumption for that standard test of urban driving conditions (UDDS drive cycle) by ~3.2%. This is attributed to smart decisions during different phases of acceleration/deceleration to switch between the electric motor and IC engine, in order to achieve the target speed.
- By training on mixed scenarios (experiment 3), the agent learns holistically and is better equipped to handle unseen scenarios for both urban and highway driving. However, the

agent performs less well on the UDDS drive cycle than an agent trained on only urban drive cycles (experiment 1).

- The trained agent does not result in large fluctuations in charge, with the battery state of charge remaining high throughout the drive cycle while still giving better fuel efficiency than the baseline.
- Using both the truncated normal distribution and shifting/scaling of NN outputs through DeepSim customizations, the agent's performance is superior to using a normal distribution. The distribution of actions taken stays skewed towards the bounds (0 and 1 here) because of clipping.

This case study acts as a proof-of-concept for improving HEV fuel efficiency using RL-agents to control the EMS. While the increases in efficiency presented here are modest, simple additions such as hyperparameter optimization, using more complex observations to define the simulation state and increasing the degree of control that the agent has over the system (i.e. a larger action space) could lead to further improvements.

Appendix

Definition of terms

Reinforcement learning

variable	**definition**
n_s	number of simulation steps between actions
dt_s	simulation time step
dt_a	action time step
a_t	action taken at time t
s_t	state at time t
r_t	reward due to taking action a_t

Hybrid Electric Vehicles

term	**definition**
HEV	Hybrid electric vehicle
ICE	Internal combustion engine
ESS	Energy storage system, eg. high voltage traction battery
EMS	Energy management system, controls the power-split ratio between ICE and electrical motor
SOC	Battery state of charge, percentage or fraction
MPG	Miles per gallon, measure of efficiency
UDDS	Urban Dynamometer Driving Schedule, set test for fuel economy based on

	urban driving conditions
HWFET	Highway Fuel Economy Test, set test for fuel economy based on highway driving conditions
US06	Drive cycle to test fuel efficiency for aggressive braking / acceleration

Simulation details

In the following section, the FASTSim simulator is introduced in more detail. Details are then given of how this can be adapted for use with reinforcement learning, using the OpenAI Gym environment as a wrapper to interact with DeepSim.

FASTSim

The original version (at https://www.nrel.gov/transportation/fastsim.html) contains four functions in a single file (*FASTSim.py*):

1. **get_standard_cycle**: user provides a drive cycle name, returns the drive cycle.
2. **get_veh**: user provides a vehicle index number, returns details on that vehicle (e.g. minimum and a maximum charge of the battery, car weight).
3. **sim_drive**: user provides the drive cycle and vehicle, which are read in using the above functions. Simulates an entire drive cycle (with charge balancing for HEVs, discussed later). There is no option to pause the simulation to take action.
4. **sim_drive_sub**: called from **sim_drive**, simulates a single run of the drive cycle. In this function, all calculations related to motor/engine power, speed/distance etc. take place. The run consists of a loop over all time steps in the drive cycle, with the various calculations relating to the HEV's drivetrain taking place within the loop.

A simulation will look like the following:

1. Initialize the simulation, using **sim_drive**, which calls **get_veh** and **get_standard_cycle**
2. Loop over all time steps in the drive cycle, which takes place in **sim_drive**
3. For each step, run the drivetrain calculations

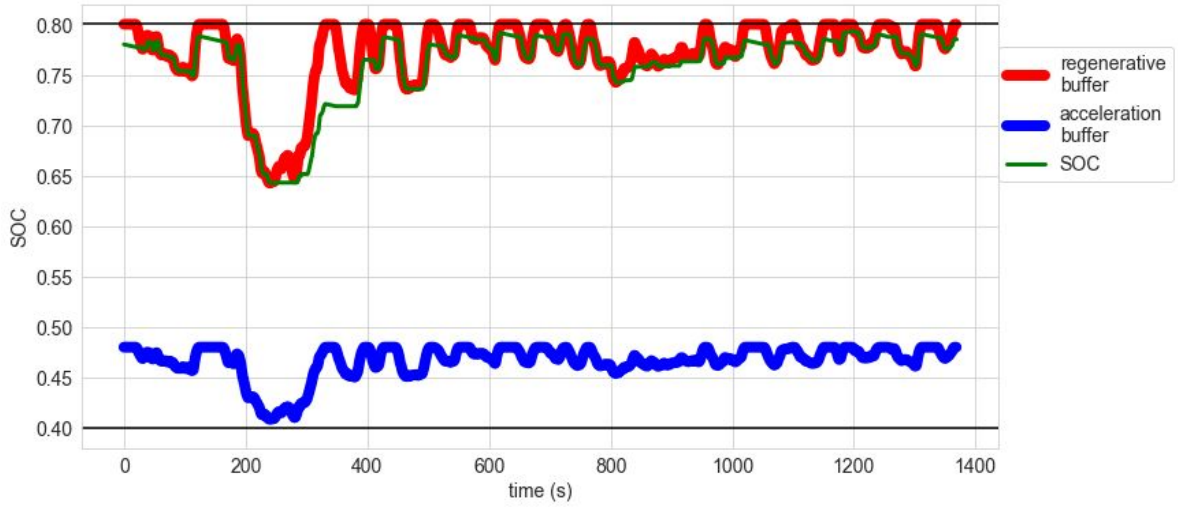
In pseudocode:

```
get_veh(vehicle_id)
get_standard_cycle(cycle_name)
for t in simulation_time_steps:
      Carry out all calculations for the drivetrain
```

```
Analyse results
```

Built-in EMS

The built-in EMS sets minimum and maximum charges for the battery and also defines buffers such that i) sufficient charge is kept in the battery to help with acceleration when required and ii) the charge is kept sufficiently below the maximum charge, so that the kinetic energy of the car can be converted and stored without breaching the maximum charge. In the plot below, the state of charge (SOC) of the car battery when using the built-in/baseline EMS is shown for the UDDS drive cycle (urban driving), along with the minimum and maximum SOC (horizontal lines at 0.4 and 0.8), and the buffers for regenerative braking and acceleration. Due to the frequent braking, the SOC remains high throughout.



For each drive cycle to be used to measure the fuel efficiency of the RL-agent controlled EMS, a baseline value is obtained by running a simulation using the default EMS.

Gym Environment

The sections below discuss how FASTSim was adapted for each of these functions

__init__

Initializes the simulation environment. This reads in a user-provided config file containing parameters for the simulation run and sets parameters that will be constant throughout all simulation runs, e.g. the paths where drive cycles and vehicle parameters are found. The function **get_veh** for FASTSim is called here, to read in the vehicle information which will be kept constant over all simulations. In addition, the action and observation spaces are set.

Action and observation spaces

These inform the training platform what types and what ranges of values it should expect for both the actions it will take and the observations it will receive as input. For example, the action to be taken is the fraction of the engine power that will be used, which is a value between 0 and 1. This is set using an OpenAI Gym spaces object, i.e.

```
self.action_space = spaces.Box(
    low=0,
    high=1,
    shape=(1,),
    dtype=np.float32,
)
```

where the user has set the minimum action to be 0, the maximum to be 1, the shape to be (1,) (a single action), and the type to be a float.

The observation space is set in a similar manner. The user must determine what variables are useful for describing the current space of the simulator. For example, the battery state of charge (SOC) and the speed of the car can be used. The SOC is a fraction of maximum charge, and since the battery can not have negative charge, ranges from 0 to 1. While a realistic range of speeds could be set here, in order to prevent errors for a parameter that can in theory take any value, it is set to range from minus to plus infinity. The observation space can then be set as

```
self.observation_space = spaces.Box(
    low=(0, -np.inf),
    high=(1, np.inf),
    shape=(2,),
    dtype=np.float32,
)
```

where the user has defined that 2 values will make up the observation (shape = (2,)), which are float values, where one ranges from 0 to 1, and the other from negative to positive infinity. Additional variables can be added to this state space e.g. the power demand to meet the desired speed, which is a float value that can run from 0 to infinity:

```
self.observation_space = spaces.Box(
    low=(0, -np.inf, 0),
    high=(1, np.inf, np.inf),
    shape=(3,),
```

```
            dtype=np.float32,
        )
```

reset

The *reset* method is used to set up everything required to run a simulation. This includes resetting any variables, counters or arrays, reading in any scenario data that changes for a new simulation run (e.g. read in a new drive cycle), and taking the simulation back to its initial state (i.e. starting at the beginning of the drive cycle). The **get_standard_cycle** function is called here to read in a drive cycle.

The method takes no arguments and returns the initial state / observation of the simulation.

step

In the *step* method, the simulator is evolved by one action timestep. The RL-agent provides an action, and the method returns the new observation / state of the simulation. The calculations carried out in the *step* method can be thought of as the contents of the outer loop of a simulation

For FASTSim, this loop was originally in the function **sim_drive_sub**, and the contents of the loop, with some modifications for passing data, can simply be copied into the *step* method.

In addition to the new state of the simulation, this method should also return:

- the reward for transitioning from the previous state to the new state.
- a flag to indicate if the simulation has finished (for FASTSim, this is simply if the car is at the end of the drive cycle).
- a dictionary containing any additional information that the user wishes to pass from the simulator.

To run a simulation as in the previous section, we can use the *reset* and *step* functions:

```
initial_observation = simulator_environment.reset()
while termination_flag is False:
      new_observation, reward, termination_flag, info_dict =
            simulator_environment.step(action)
Analyse results
```

Configuration parameters

Below are listed some of the important configuration parameters used by Ray RLLib and FastSim along with short descriptions of their purpose.

RL Agent configuration

Field	Description	Value used
`rollout_fragment_length`	Divide episodes into fragments of this many steps during rollouts for training	300
`train_batch_size`	Number of episode steps formed by concatenating fragments of size rollout_fragment_length sent to SGD for training	30000
`sgd_minibatch_size`	Number of episode steps used in a single SGD update	300
`num_sgd_iter`	SGD update iterations performed over the same training batch	80
`gamma`	Factor for discounting future rewards till the episode end	1.0
`lambda`	Factor to scale the discount factor in estimating advantage	1.0
`entropy_coeff`	Coefficient of entropy in generalized loss calculation	1e-6
`kl_target`	Max allowed KL divergence between consecutive policies after update	0.02
`clip_param`	Clipping parameter ϵ for PPO surrogate objective	0.2
`fcnet_hiddens`	Sizes of fully connected hidden layers	256, 256, 64
`init_std`	Initial standard deviation to scale the output deviation	0.175
`free_log_std`	Set log std. deviation output independent of network states (only depends on bias)	True

FastSim Environment configuration

Field	Description	Value used (training & evaluation)

		Exp. 1 and 3	**Exp. 2**
`break_up_cycle`	Randomly stop drive cycles i.e. end episode, minimum time of 300s per episode	True	True
`charge_ balancing`	Balance charge over the drive cycle	False (in eval=True)	False (in eval=False)
`delta_action`	Use change in action, rather than the action	False	False
`discrete_ actions`	Use discrete actions instead of continuous	False	False
`initial_soc`	Initial state-of-charge (-1 for mean of max and min, 0 for random SOC)	0.79	0 (in_eval = {0.40, 0.45, 0.50...0.80})
`num_state_ stack`	Number of previous states to use as observations	6	6
`observation_ list`	List of variables to use as state observation	soc, power_demand, speed, speed_change	soc, power_demand, speed, speed_change
`reward_scale_ fac`	Prefactor to scale the reward by	1000	1000
`use_final_soc`	To start next drive cycle from the final SOC of the previous	True	False