fleetrl

Release 1.0.1

Enzo Alexander Cording

May 05, 2024

USER GUIDE

Github: https://github.com/EnzoCording/FleetRL

CHAPTER

ONE

SOFTWARE FEATURES:

- Base-derived class architecture and easily interchangeable configurations
- Modular implementation approach for easily extendable model and methods
- PEP8 compliant (unified code style)
- Documented functions and classes
- Stable-Baselines3 integration, parallelization, tensorboard support
- Extensive data logging and evaluation possibilities

CHAPTER

TWO

UNIQUE IMPLEMENTATIONS:

- Non-linear battery degradation
- Fleet schedule generation - inspired by emobpy and applied to commercial fleets
- Bi-directional charging
- Building load, grid connection limit, PV, spot price
- Arbitrarily variable episode length, 15-min time resolution
- Benchmarking of RL methods with static benchmarks: uncontrolled charging, night charging, distributed charging
- **Benchmarking of RL methods with linear optimization**

2.1 Overview

FleetRL is a Reinforcement Learning (RL) environment for EV charging optimization with a special focus on commercial vehicle fleets. Its main function is the modelling of real-world charging processes. FleetRL was developed with a modular approach, keeping in mind that improvements and additions are important to maintain the value of this framework. Emphasis was therefore laid on readability, ease of use and ease of maintenance. For example, the base-derived class architecture was used throughout the framework, allowing for easy interchangeability of modules such as battery degradation.

Emphasis was also laid on customizability: own schedules can be generated, electricity prices can be switched by changing a csv file, episode length, time resolution, and EV charging-specific parameters can be changed in their respective config files.

Physical model

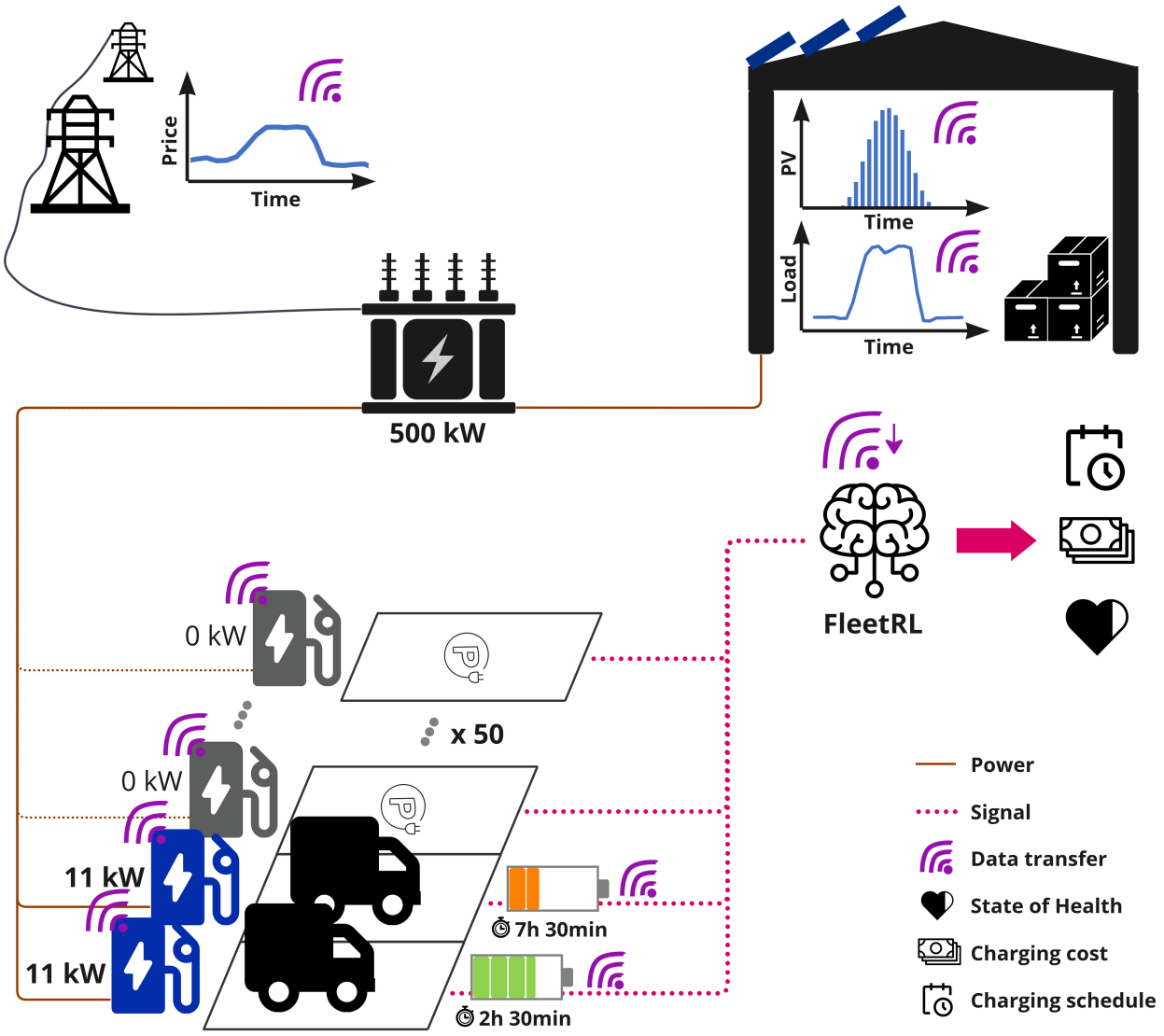
As seen in the graphic below, the physical scope currently includes the EVs, the building (load + PV), a limited grid connection, and electricity prices (both for purchase and feed-in).

The EVs have probabilistic schedules that are generated beforehand. The main objective is thereby to minimize charging cost while respecting the constraints of the schedules, grid connection and SOC requirements. Battery degradation is modelled, both linearly and non-linearly. Electricity prices are taken from the EPEX spot market. PV production data is taken from the MERRA-2 open data set, available here. Building load was taken from the NREL TMY-3 dataset.

Some assumptions are taken to limit the degree of complexity of the optimization problem:

- All connections are three-phase, and load imbalance is not considered.
- Only active power is considered when calculating transformer loading. Capacitive and inductive properties of electrical equipment are not considered.

- Although bidirectional charging is enabled, it only allows for energy arbitrage. Frequency regulation is not implemented in this study.
- The companies are modelled as price takers and are assumed not to influence the electricity price with their demand.
- Battery degradation is modelled non-linearly, taking into account rainflow cycle-counting and SEI-film formation according to Xu et al..



Code structure

FleetRL is based on the OpenAI / farama foundation gym framework and "basically" implements the step, reset, and init functions. To train RL agents, the stable-baselines3 framework is used due to its high-quality implementations, level of maintenance and documentation. Different agents can be plugged into the environment plug-and-play, only having to change a few lines of code.

To train RL agents on a FleetRL environment, an env object needs to be created, which inherits from gym and implements the necessary methods and the entire logic of the EV charging problem. The implementation can be found under `FleetRL.fleet_env.fleet_environment`. As can be seen, functions such as EV charging, or calculating battery

degradation are outsourced to separate modules to maintain readability. This way, a submodule can also be changed without having to touch the main logic of the code.

Leveraging base-derived class architecture

FleetRL makes use of parent classes and sub-classes that implement the methods of the parent class - this is also known as a base-derived hierarchy. For example, FleetRL uses `time_picker` to pick a starting date for a new episode. This can either be random, or always the same date. A parent class `TimePicker` thereby exists:

```
class TimePicker:
    def choose_time(self, db: pd.Series, freq: str, end_cutoff: int) -> Timestamp:
        """
        Parent class for time picker objects
        :param db: dataframe from env
        :param freq: frequency specification string for pandas
        :param end_cutoff: amount of days cut off at the end to allow some buffer. In␣
↪the eval time picker case,
        the end cutoff specifies the size of the validation set.
        :return: A chosen time stamp
        """
        raise NotImplementedError("This is an abstract class.")
```

As can be seen, it dictates the inputs and outputs, as well as the methods the class contains -> these must be matching exactly when implementing a sub-class of `TimePicker`. The parent class methods cannot be called directly, this would raise a `NotImplementedError`.

A sub-class is implemented as follows, taking the example of the `StaticTimePicker`:

```
from FleetRL.utils.time_picker.time_picker import TimePicker

class StaticTimePicker(TimePicker):
    """
    Picks a static / always the same starting time.
    """

    def __init__(self, start_time: str = "01/01/2020 15:00"):
        """
        :param start_time: When initialised, start time is specified
        """
        self.start_time = start_time

    def choose_time(self, db: pd.Series, freq: str, end_cutoff: int) -> Timestamp:
        chosen_start_time = pd.to_datetime(self.start_time)
        # return start time
        return chosen_start_time
```

As can be seen, `StaticTimePicker` inherits from `TimePicker`, and implements its methods. In the code, an object of the type `TimePicker` can then be created as follows:

```
tp: TimePicker = StaticTimePicker()
```

If a different time picker should be chosen, it can be changed by changing one line of code. In this case, the `RandomTimePicker` is chosen instead:

```
tp: TimePicker = RandomTimePicker()
```

Note: This is already automated in FleetRL, and the right time picker module can be specified via a string in the input parameters when creating an env object.

Note: When writing an own sub-class, it must be ensured that all methods are implemented, and that the methods follow the same inputs/outputs as the parent class. Once this is done, the own implementation can be used in FleetRL by changing one line of code, as shown above.

2.2 Installation

Prerequisites

- FleetRL requires Python >=3.8

Note: Python >= 3.10 is strongly recommended.

Note: The creation of a virtual environment is strongly recommended. To be able to use GPU compute, CUDA drivers must be installed (11.8 was mostly used during development).

Installation via Github repository:

- Unzip the package
- Rename directory from FleetRL-master to FleetRL
- cd into /FleetRL
- pip install -r requirements.txt

Note: On remote environments on vast.ai it can be necessary to run `pip install -U numpy` prior to installing FleetRL

Miniconda Windows

In this example, FleetRL can be installed completely from scratch, only Miniconda is required. Run the commands below consecutively.

```
conda create -n **environment_name** python=3.10
conda activate **environment_name**
pip install jupyter
jupyter notebook
```

Inside the Jupyter Notebook, being in the FleetRL directory:

```
!pip install -r requirements.txt
# restart kernel
import FleetRL
```

At this point, the `complete_pipeline.ipynb` should run completely. To use GPU, CUDA must be properly configured.

2.3 Creating a custom environment

This section walks through the process of creating a completely custom-made environment, potentially to model your own commercial use-case and train an RL agent on it.

Note: Emphasis was laid on keeping the amount of coding to a minimum for running a custom use-case.

The different components

To build a custom use-case in FleetRL, an understanding of the different building blocks of the framework is required. Inputs come in different formats, e.g. time-series, static inputs, and config files. Below, each component is explained and how to modify it.

2.3.1 Time-series

Vehicle schedules

Vehicle schedules are the most important time-series required in FleetRL. They are saved as csv files. Agents can be trained solely with schedules, if price, PV, and building load are set to be ignored. The schedule format is adapted from emobpy, an open-source framework for EV schedule generation, to allow for cross-compatibility with emobpy outputs. The following columns are required:

- **date**: The "date" column displays the date, preferably in the format `YYYY-MM-dd hh:mm:ss`. By default, the time resolution is 15 minutes in the schedules - a lower time resolution can later be adjusted in the settings.
- **Distance_km**: Distance travelled in each time step, in km
- **Consumption_kWh**: Energy consumed in each time step, in kWh
- **Location**: Location specifier. "home" and "driving" must always be included.
- **ChargingStation**: Specifying the type of charging station. "home" and "none" must always be included. A ChargingStation of "none" means that the car is "driving".
- **ID**: The vehicle ID. Starts with 0 and goes until (N-1), if N cars are present in the schedule.
- **PowerRating_kW**: The amount of kW at the charger.

Note: The "date" column in the schedule csv file dictates the timeframe of the entire simulation.

Note: Custom schedules can be generated within FleetRL. It currently is possible to generate two mobility scenarios: departure - arrival, and departure - break (arrival) - break (departure) - arrival. A fully automated process exists, which is explained in the next section - custom schedules are thereby generated when creating the environment object for the first time. To customize the statistics of the vehicle schedules, the `schedule_config.py` file must be modified. It specifies the behavioral statistics: arrival/departure times (mean, standard dev), energy consumption (mean, standard dev), distance (mean, standard dev), and differences between weekdays, saturday, and sunday.

Electricity prices

Electricity prices must be csv files and have the same date range as the vehicle schedules. The backend of FleetRL is currently set to German spot prices that can be downloaded from smard.de or the Entso-E transparency platform. The following columns are required:

- **date**: Date, preferably in `YYYY-MM-dd hh:mm:ss` format. If time resolutions of lower than 15 minutes are provided, FleetRL automatically upsamples the data to match the time resolution of the vehicle schedules. German spot prices have a time resolution of 1 hour and are thus upscaled to 15 minutes.
- **Deutschland/Luxemburg [€/MWh] Original resolutions**: This is the identifier for the German/Lux electricity market on ENTSO-E. The name of the price column can also be changed in the `data_processing.load_prices` method.

Note: A csv file for buying ("price") and selling ("tariff") is required. The tariff csv must have a "tariff" column, where the prices are specified.

Building load

The TMY-3 dataset was used for building load time series. Different commercial buildings exist, such as "warehouse", "office", "hospital", etc. The state of Washington was used due to its resemblance with the German climate. Following columns must be present:

- **date**: Preferably `YYYY-MM-dd hh:mm:ss` format, same range as the vehicle schedules. Varying time resolutions are automatically adjusted.
- **load**: The "load" column specifies the load in kW in the current time step
- **pv**: PV generation can be specified in a "pv" column. Alternatively, PV generation can be input via a separate CSV file. In that case, it must have a "date" and "pv" column.

2.3.2 Static inputs

Use-case-specific inputs

- Number of vehicles: This is specified via the number of vehicles present in the schedule file (taking the max of the "ID" column).
- Vehicle type: Battery capacity must be specified in `fleet_environment.__init__` and `episode.load_calculation`. In `config.ev_config`, further parameters can be changed, such as onboard charger capacity, charging efficiency, etc. EVSE (EV charger) capacity is specified under `episode.load_calculation`.
- Grid connection: The grid connection is specified in `episode.load_calculation`. It can either be hard-coded, or adjusted depending on the number of cars and the maximum building load (e.g. last-mile delivery).
- Electricity tax, markups, fees: Can be adjusted when creating an env object, or under `config.ev_config`.

Episode-specific inputs

- Episode length: In hours, can be adjusted when creating an env object.
- Time resolution: Hourly ("1H") or 15 minutes ("15T"). Can be set under `config.time_config`.
- Auxiliary inputs: Can help the agent to solve the task by providing additional information and preprocesses metrics.

Reward function and penalties

`config.score_config` specifies the shape, scale and type of rewards and penalties. A reward signal exists for the following:

- Charging expenses: The agent receives a reward signal based on the money spent on EV charging
- Overloading the grid connection: A penalty signal is sent if the grid connection exceeds the nominal trafo rating. Its shape follows the sigmoid function: small exceeding values (0-10%) can be handled by a trafo and thus do not require a strong penalty. Higher values (15-50%) must be avoided and thus require strong penalties. Everything

above most likely represents a system failure, and a further increase does not worsen the situation - the level of penalties thus flattens off towards high overloadings.

- SOC violation: If a car leaves without being fully charged, a similarly shaped penalty signal is sent to the agent. It too is sigmoid shaped.
- Battery overcharging: If the agent sends a signal that would overcharge / over-discharge the battery, a penalty signal can be sent.
- Invalid action: If a charging signal is sent to an empty charging spot, a penalty can be sent to the agent.

2.3.3 Summary

When creating a custom commercial use-case, do the following:

Environment object constructor

Customizable parameters can be found below:

```
# environment arguments - adjust settings if necessary
# additional settings can be changed in the config files
env_kwargs = {"schedule_name": str(n_evs) + "_" + str(use_case) + ".csv",  # schedule␣
↪name
              "building_name": "load_" + str(use_case) + ".csv",  # name of building␣
↪load (and pv) time series
              "use_case": use_case,  # put "custom" here, or your own identifier (code␣
↪modifications necessary)
              "include_building": True,  # including building load in the observations
              "include_pv": True,  # including PV in the observations
              "time_picker": "random",  # randomly picking a starting time from the␣
↪training set
              "deg_emp": False,  # not using empirical degradation (uses non-linear␣
↪degradation model)
              "include_price": True,  # including prices in the observations
              "ignore_price_reward": False,  # True: not sending charging cost reward␣
↪signals to the agent
              "ignore_invalid_penalty": False,  # True: not sending invalid action␣
↪penalties to the agent
              "ignore_overcharging_penalty": False,  # True not sending overcharging␣
↪penalties to the agent
              "ignore_overloading_penalty": False,  # True not sending overloading␣
↪penalties to the agent
              "episode_length": n_train_steps,  # Episode length in hours
              "normalize_in_env": norm_obs_in_env,  # Normalize observations within␣
↪FleetRL (min/max normalization)
              "verbose": 0,  # print statements, can slow down FPS
              "aux": True,  # Include auxiliary information in observation
              "log_data": False,  # log data in a dataframe (best used in evaluation, no␣
↪need during training)
              "calculate_degradation": True,  # call the degradation class (can slow␣
↪down FPS)
              "target_soc": 0.85,  # target SOC
              "gen_schedule": gen_new_schedule,  # generate a new schedule upon env␣
↪object creation
              "gen_start_date": "2022-01-01 00:00",  # start date of the new schedule
```

(continues on next page)

(continued from previous page)

```
            "gen_end_date": "2022-12-31 23:59:59",  # end date of the new schedule
            "gen_name": "my_sched.csv",  # name of the generated schedule
            "gen_n_evs": 1,  # number of EVs to be generated (approx. 20 min. per EV)
            "seed": 42  # RNG seed for the generation
            }

if scenario == "tariff":  # if tariff is chosen, price and tariff are different, there␣
↪are mark-ups and deductions
    env_kwargs["spot_markup"] = 10
    env_kwargs["spot_mul"] = 1.5
    env_kwargs["feed_in_ded"] = 0.25
    env_kwargs["price_name"] = "spot_2021_new.csv"
    env_kwargs["tariff_name"] = "fixed_feed_in.csv"
elif scenario == "arb":  # if arb is chosen, price and tariff are the same, no mark-ups␣
↪or deductions
    env_kwargs["spot_markup"] = 0
    env_kwargs["spot_mul"] = 1
    env_kwargs["feed_in_ded"] = 0
    env_kwargs["price_name"] = "spot_2021_new.csv"
    env_kwargs["tariff_name"] = "spot_2021_new_tariff.csv"
```

Schedule config

Configure the statistics in `schedule.schedule_config`. View the "custom" case.

Load calculation module

Configure the battery capacity, EVSE power and grid limit in `load_calculation.load_calculation`. View the custom case.

Other configs

Modify the parameters in `fleet_env.config.ev_config`, `fleet_env.config.score_config`, and `fleet_env.config.time_config`.

2.4 Agent Training

This section walks through the process of creating a FleetRL env object, and training an RL agent on it. The creation of the environment walks through important parameters, generating custom schedules, and additional parameters that can be used to customize the environment.

For training, stable-baselines3 is used, in this case the PPO agent. Progress is logged via tensorboard, as well as a progress bar. Agent artifacts are saved, so they can be reused at a different point in time or on another machine. Note that it is possible to create wandb callbacks with SB3, which enable advanced logging and progress visualization, as well as automatic storing of model artifacts.

Importing dependencies

```
import datetime as dt
import numpy as np
import math
import matplotlib.pyplot as plt
from typing import Literal
import pandas as pd
```

(continues on next page)

(continued from previous page)

```
import time
import os

from FleetRL.fleet_env import FleetEnv

from stable_baselines3.common.vec_env import VecNormalize, SubprocVecEnv
from stable_baselines3.common.env_util import make_vec_env
from stable_baselines3 import PPO
from stable_baselines3.common.callbacks import EvalCallback, ProgressBarCallback,␣
↪BaseCallback
from stable_baselines3.common.logger import HParam

from pink import PinkActionNoise
from stable_baselines3.common.noise import OrnsteinUhlenbeckActionNoise,␣
↪NormalActionNoise
```

Configuring the environment

Below, a basic configuration is shown for a last-mile delivery use-case. It features multiple EVs and multiple parallel environments. New schedules will be generated in this case - this can be changed by setting `gen_new_sched` and `gen_new_test_sched` to False.

```
# define fundamental parameters
run_name = "LMD_2022_arbitrage_PPO"
n_train_steps = 48  # number of hours in a training episode
n_eval_steps = 8600  # number of hours in one evaluation episode
n_eval_episodes = 1  # number of episodes for evaluation
n_evs = 2  # number of evs
n_envs = 2  # number of envs parallel - has to be equal to 1, if train_freq = (1,␣
↪episode) or default setting
time_steps_per_hour = 4  # temporal resolution
use_case: str = "lmd"  # for file name
scenario: Literal["arb", "tariff"] = "tariff"  # arbitrage or tariff
gen_new_schedule = True  # generate a new schedule - refer to schedule generator and its␣
↪config to change settings
gen_new_test_schedule = True  # generate a new schedule for agent testing
```

Training parameters change settings in the backends of FleetRL and stable-baselines3. This includes normalization, vectorization of environments, total training steps and the interval at which models are saved.

```
# training parameters
norm_obs_in_env = False  # normalize observations in FleetRL (max, min normalization)
vec_norm_obs = True  # normalize observations in SB3 (rolling normalization)
vec_norm_rew = True  # normalize rewards in SB3 (rolling normalization)
total_steps = int(1e6)  # total training time steps
saving_interval = 50000  # interval for saving the model
```

Creating an environment object

To instantiate a FleetEnv object, some arguments are required, and some are optional. Depending on the pricing scenario, the right input files and markups are chosen. Depending on the use-case, the schedule and building load input files are chosen.

```
# environment arguments - adjust settings if necessary
# additional settings can be changed in the config files
env_kwargs = {"schedule_name": str(n_evs) + "_" + str(use_case) + ".csv",
              "building_name": "load_" + str(use_case) + ".csv",
              "use_case": use_case,
              "include_building": True,
              "include_pv": True,
              "time_picker": "random",
              "deg_emp": False,
              "include_price": True,
              "ignore_price_reward": False,
              "ignore_invalid_penalty": False,
              "ignore_overcharging_penalty": False,
              "ignore_overloading_penalty": False,
              "episode_length": n_train_steps,
              "normalize_in_env": norm_obs_in_env,
              "verbose": 0,
              "aux": True,
              "log_data": False,
              "calculate_degradation": True,
              "target_soc": 0.85,
              "gen_schedule": gen_new_schedule,
              "gen_start_date": "2022-01-01 00:00",
              "gen_end_date": "2022-12-31 23:59:59",
              "gen_name": "my_sched.csv",
              "gen_n_evs": 1,
              "seed": 42
              }

if scenario == "tariff":
    env_kwargs["spot_markup"] = 10
    env_kwargs["spot_mul"] = 1.5
    env_kwargs["feed_in_ded"] = 0.25
    env_kwargs["price_name"] = "spot_2021_new.csv"
    env_kwargs["tariff_name"] = "fixed_feed_in.csv"
elif scenario == "arb":
    env_kwargs["spot_markup"] = 0
    env_kwargs["spot_mul"] = 1
    env_kwargs["feed_in_ded"] = 0
    env_kwargs["price_name"] = "spot_2021_new.csv"
    env_kwargs["tariff_name"] = "spot_2021_new_tariff.csv"
```

Create environments

Vectorized environments are created via the respective SB3 method. The SubprocVecEnv is used because it allows for parallel processing (unlike `DummyVecEnv`). The VecEnv is wrapped in a normalization wrapper via `VecNormalize`.

```
train_vec_env = make_vec_env(FleetEnv,
                             n_envs=n_envs,
                             vec_env_cls=SubprocVecEnv,
                             env_kwargs=env_kwargs,
                             seed = env_kwargs["seed"])
```

(continues on next page)

(continued from previous page)

```
train_norm_vec_env = VecNormalize(venv=train_vec_env,
                                  norm_obs=vec_norm_obs,
                                  norm_reward=vec_norm_rew,
                                  training=True,
                                  clip_reward=10.0)
```

For the validation environment, a new schedule does not need to be generated. However, the eval time picker needs to be used, and the schedule name of the training environment needs to be adopted.

```
env_kwargs["time_picker"] = "eval"
env_kwargs["gen_schedule"] = False
env_kwargs["schedule_name"] = env_kwargs["gen_name"]

eval_vec_env = make_vec_env(FleetEnv,
                            n_envs=n_envs,
                            vec_env_cls=SubprocVecEnv,
                            env_kwargs=env_kwargs,
                            seed=env_kwargs["seed"])

eval_norm_vec_env = VecNormalize(venv=eval_vec_env,
                                 norm_obs=vec_norm_obs,
                                 norm_reward=vec_norm_rew,
                                 training=True,
                                 clip_reward=10.0)
```

Adding callbacks

The SB3 EvalCallback is used to run a validation on the validation set. This tests the currently training agent on a separate portion of the dataset, allowing for better judgement of performance on unseen data.

```
eval_callback = EvalCallback(eval_env=eval_norm_vec_env,
                             warn=True,
                             verbose=1,
                             deterministic=True,
                             eval_freq=max(10000 // n_envs, 1),
                             n_eval_episodes=5,
                             render=False,
                             )
```

The HyperParamCallback is used to log metric in TensorBoard. Depending on the used RL algorithm, more parameters can be logged.

```
class HyperParamCallback(BaseCallback):
    """
    Saves hyperparameters and metrics at start of training, logging to tensorboard
    """

    def _on_training_start(self) -> None:
        hparam_dict = {
            "algorithm": self.model.__class__.__name__,
            "learning rate": self.model.learning_rate,
            "gamma": self.model.gamma,
        }
```

(continues on next page)

(continued from previous page)

```
        metric_dict = {
            "rollout/ep_len_mean": 0,
            "train/value_loss": 0.0,
        }

        self.logger.record(
            "hparams",
            HParam(hparam_dict, metric_dict),
            exclude=("stdout", "log", "json", "csv")
        )

    def _on_step(self) -> bool:
        return True

hyperparameter_callback = HyperParamCallback()
```

A progress bar can be included. This might not show during live training in some remote computing environments.

```
progress_bar = ProgressBarCallback()
```

Note: A wandb callback exists for SB3. Check the wandb documentation for implementation.

Adding action noise, e.g. for TD3 and DDPG

```
n_actions = train_norm_vec_env.action_space.shape[-1]
param_noise = None
noise_scale = 0.1
seq_len = n_train_steps * time_steps_per_hour
action_noise = PinkActionNoise(noise_scale, seq_len, n_actions)
```

Note: `PinkActionNoise` was used here, according to https://openreview.net/pdf?id=hQ9V5QN27eS. Alternatively, `NormalActionNoise`, or `OrnsteinUhlenbeckNoise` of the SB3 library can be used.

Instantiating model

To avoid specific model tuning for each new use-case, it is recommended to first try a model's default parameters. Below, a hyperparameter configuration is proposed for PPO that performed well for all three use-cases. It can potentially yield potential performance increases.

```
model = PPO(policy="MlpPolicy",
            verbose=1, # setting verbose to 0 can introduce performance increases in␣
↪jupyterlab environments
            env=train_norm_vec_env,
            tensorboard_log="./tb_log")

# might introduce performance increases
            # gamma=0.99,
            # learning_rate=0.0005,
            # batch_size=128,
```

(continues on next page)

(continued from previous page)

```
            # n_epochs=8,
            # gae_lambda=0.9,
            # clip_range=0.2,
            # clip_range_vf=None,
            # normalize_advantage=True,
            # ent_coef=0.0008,
            # vf_coef=0.5,
            # max_grad_norm=0.5,
            # n_steps=2048)
```

Creating tensorboard instance. Port can be specified in case a certain port is free on remote computing environments. `bind_all` might be required by some remote machines.

```
%reload_ext tensorboard
%tensorboard --logdir ./tb_log --bind_all --port 6006
```

Setting directories according to path names.

```
comment = run_name
time_now = int(time.time())
trained_agents_dir = f"./RL_agents/trained_agents/vec_PPO_{time_now}_{run_name}"
logs_dir = f"./RL_agents/trained_agents/logs/vec_PPO_{time_now}_{run_name}"

if not os.path.exists(trained_agents_dir):
    os.makedirs(trained_agents_dir)

if not os.path.exists(logs_dir):
    os.makedirs(logs_dir)
```

Model training

Per interval, a unique model artifact is saved. Additionally, an artifact is saved regarding the normalization metrics and the agent model - this is overwritten each time. The instantiated callbacks must be included here.

```
# model training
# models are saved in a specified interval: once with unique step identifiers
# model and the normalization metrics are saved as well, overwriting the previous file␣
↪every time
for i in range(0, int(total_steps / saving_interval)):
    model.learn(total_timesteps=saving_interval,
                reset_num_timesteps=False,
                tb_log_name=f"PPO_{time_now}_{comment}",
                callback=[eval_callback, hyperparameter_callback, progress_bar])

    model.save(f"{trained_agents_dir}/{saving_interval * i}")

    # Don't forget to save the VecNormalize statistics when saving the agent
    log_dir = "./RL_agents/trained_agents/tmp/vec_PPO/"
    model.save(log_dir + f"PPO-fleet_{comment}_{time_now}")
    stats_path = os.path.join(log_dir, f"vec_normalize-{comment}_{time_now}.pkl")
    train_norm_vec_env.save(stats_path)
```

2.5 Agent Evaluation

Agent evaluation requires a trained RL agent, in form of a .zip artifact generated from SB3. The evaluation features a comparison with uncontrolled charging to allow for a first basic comparison. It is optional and can be toggled off to save compute time.

Import requirements

```
import datetime

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3 import TD3, PPO
from stable_baselines3.common.vec_env import VecNormalize, SubprocVecEnv
from stable_baselines3.common.env_util import make_vec_env
from FleetRL.fleet_env.fleet_environment import FleetEnv
```

Multi-processing wrapper

The evaluation uses vectorized environments that allow for parallelization. If this script is run in a Jupyter notebook, no changes need to be made. If, however, the script is run as a .py file, it needs to be wrapped as follows:

```
if __name__ == "__main__":
    # code here
```

Defining fundamental parameters

By default, `n_steps` is set to 8600. This means that the evaluation episode is set to 8600 hours. The trained agent is therefore tested on one year of unseen schedule data. A separately generated schedule is used that the agent did not see during training.

```
# define parameters here for easier change
n_steps = 8600
n_episodes = 1
n_evs = 1
n_envs = 1
file_name_comment = "comment"  # added to log pickle file names
```

Environment creation

The testing environment is created. The parameters are the same as for training - only the schedule differs: `lmd_sched_single_eval.csv`. The same normalization and vectorization is performed. Generally, agents are cross-compatible with environments of same dimension and boundaries. A 1-car agent trained on the environment for caretakers can thus be used on the last-mile delivery environment, if the `gym.Spaces` bounds are the same. If normalization is conducted in FleetRL, the bounds are [0,1]. If no normalization is conducted, the bounds are [-inf, inf]. This ensures maximum cross-compatibility. Similarly, the environment for the uncontrolled charging agent is created (`dumb_vec_env, dumb_norm_vec_env`).

```
# make env for the agent
eval_vec_env = make_vec_env(FleetEnv,
                            n_envs=n_envs,
                            vec_env_cls=SubprocVecEnv,
                            seed=0,
```

(continues on next page)

(continued from previous page)

```
                                env_kwargs={
                                    "schedule_name": "lmd_sched_single_eval.csv",  #␣
↪separate testing schedule
                                    "building_name": "load_lmd.csv",
                                    "price_name": "spot_2021_new.csv",
                                    "tariff_name": "spot_2021_new_tariff.csv",
                                    "use_case": "lmd",
                                    "include_building": True,
                                    "include_pv": True,
                                    "time_picker": "static",
                                    "deg_emp": False,
                                    "include_price": True,
                                    "ignore_price_reward": False,
                                    "ignore_invalid_penalty": False,
                                    "ignore_overcharging_penalty": False,
                                    "ignore_overloading_penalty": False,
                                    "episode_length": n_steps,
                                    "normalize_in_env": False,
                                    "verbose": 0,
                                    "aux": True,
                                    "log_data": True,
                                    "calculate_degradation": True,
                                    "spot_markup": 0,
                                    "spot_mul": 1,
                                    "feed_in_ded": 0
                                })

eval_norm_vec_env = VecNormalize(venv=eval_vec_env,
                                 norm_obs=True,
                                 norm_reward=True,
                                 training=True,
                                 clip_reward=10.0)

dumb_vec_env = make_vec_env(FleetEnv,
                            n_envs=n_envs,
                            vec_env_cls=SubprocVecEnv,
                            seed=0,
                            env_kwargs={
                                "schedule_name": "lmd_sched_single_eval.csv",
                                "building_name": "load_lmd.csv",
                                "price_name": "spot_2021_new.csv",
                                "tariff_name": "spot_2021_new_tariff.csv",
                                "use_case": "lmd",
                                "include_building": True,
                                "include_pv": True,
                                "time_picker": "static",
                                "deg_emp": False,
                                "include_price": True,
                                "ignore_price_reward": False,
                                "ignore_invalid_penalty": False,
                                "ignore_overcharging_penalty": False,
                                "ignore_overloading_penalty": False,
```

(continues on next page)

(continued from previous page)

```
                                "episode_length": n_steps,
                                "normalize_in_env": False,
                                "verbose": 0,
                                "aux": True,
                                "log_data": True,
                                "calculate_degradation": True,
                                "spot_markup": 0,
                                "spot_mul": 1,
                                "feed_in_ded": 0
                            })

dumb_norm_vec_env = VecNormalize(venv=dumb_vec_env,
                                 norm_obs=True,
                                 norm_reward=True,
                                 training=True,
                                 clip_reward=10.0)
```

Loading models

The normalization metrics can be loaded via `VecEnv.load(load_path, venv)`. This is optional. The RL agent is loaded. The path to the .zip artifact and the environment must be specified. Optionally, a custom_objects parameter can be parsed to make sure that observation and action space are correctly configured.

```
eval_norm_vec_env.load(load_path="./tmp/vec_PPO/vec_normalize-LMD_2021_arbitrage_PPO_
↪mul3.pkl", venv=eval_norm_vec_env)
model = PPO.load("./tmp/vec_PPO/PPO-fleet_LMD_2021_arbitrage_PPO_mul3.zip", env = eval_
↪norm_vec_env,
                 custom_objects={"observation_space": eval_norm_vec_env.observation_space,
                                 "action_space": eval_norm_vec_env.action_space})
```

RL agent evaluation

Agents are evaluated via `evaluate_policy`. The model, the environment, the number of episodes and the deterministic flag are parsed. `deterministic=True` ensures that several evaluations of the same agents yield the same results - ensuring reproducibility. Random fluctuations due to random number generators or statistical distributions are eliminated.

```
mean_reward, _ = evaluate_policy(model, eval_norm_vec_env, n_eval_episodes=n_episodes,␣
↪deterministic=True)
print(mean_reward)
```

Once `evaluate_policy` concluded, the environment stepped through 8600 hours. Meanwhile, FleetRL logged every important metric, allowing for post-processing and thorough analyses. These can be accessed via `env_method("get_log")[0]`, as shown below.

```
log_RL = model.env.env_method("get_log")[0]
```

Uncontrolled charging agent

The start time of the evaluation environment is extracted and set as start time for the uncontrolled charging environment. The environment is then stepped through for the same amount of time steps and the log is extracted.

```
# start date extraction and setting the same date to the uncontrolled charging env
rl_start_time = model.env.env_method("get_start_time")[0]
```

(continues on next page)

(continued from previous page)

```
dumb_norm_vec_env.env_method("set_start_time", rl_start_time)

print("################################################################")

episode_length = n_steps
timesteps_per_hour = 4
n_episodes = n_episodes
dumb_norm_vec_env.reset()

# uncontrolled charging agent: action of "1" is sent for each time step -> charging␣
↪immediately upon arrival
for i in range(episode_length * timesteps_per_hour * n_episodes):
    if dumb_norm_vec_env.env_method("is_done")[0]:
        dumb_norm_vec_env.reset()
    dumb_norm_vec_env.step([np.ones(n_evs)])

# log extraction from the vec_env
dumb_log = dumb_norm_vec_env.env_method("get_log")[0]
```

Post-processing

Once both agents ran in the environments and the logs have been extracted, they can be used to extract useful information on charging expenses, state of health, violations, rewards, etc.

```
# index reset and the last row of the dataframe is removed
log_RL.reset_index(drop=True, inplace=True)
log_RL = log_RL.iloc[0:-2]
dumb_log.reset_index(drop=True, inplace=True)
dumb_log = dumb_log.iloc[0:-2]

# computing key performance metrics
rl_cashflow = log_RL["Cashflow"].sum()
rl_reward = log_RL["Reward"].sum()
rl_deg = log_RL["Degradation"].sum()
rl_overloading = log_RL["Grid overloading"].sum()
rl_soc_violation = log_RL["SOC violation"].sum()
rl_n_violations = log_RL[log_RL["SOC violation"] > 0]["SOC violation"].size
rl_soh = log_RL["SOH"].iloc[-1]

dumb_cashflow = dumb_log["Cashflow"].sum()
dumb_reward = dumb_log["Reward"].sum()
dumb_deg = dumb_log["Degradation"].sum()
dumb_overloading = dumb_log["Grid overloading"].sum()
dumb_soc_violation = dumb_log["SOC violation"].sum()
dumb_n_violations = dumb_log[dumb_log["SOC violation"] > 0]["SOC violation"].size
dumb_soh = dumb_log["SOH"].iloc[-1]

print(f"RL reward: {rl_reward}")
print(f"DC reward: {dumb_reward}")
print(f"RL cashflow: {rl_cashflow}")
print(f"DC cashflow: {dumb_cashflow}")

total_results = pd.DataFrame()
```

(continues on next page)

(continued from previous page)

```
total_results["Category"] = ["Reward", "Cashflow", "Average degradation per EV",
↪"Overloading", "SOC violation", "# Violations", "SOH"]

total_results["RL-based charging"] = [rl_reward,
                                      rl_cashflow,
                                      np.round(np.mean(rl_deg), 5),
                                      rl_overloading,
                                      rl_soc_violation,
                                      rl_n_violations,
                                      np.round(np.mean(rl_soh), 5)]

total_results["Dumb charging"] = [dumb_reward,
                                  dumb_cashflow,
                                  np.round(np.mean(dumb_deg), 5),
                                  dumb_overloading,
                                  dumb_soc_violation,
                                  dumb_n_violations,
                                  np.round(np.mean(dumb_soh), 5)]

print(total_results)
```

Plotting

As an example, the charging strategies of the RL agent and the uncontrolled charging strategy are plotted - the mean of each quarter hour is plotted, indicating when charging signals are sent to the battery.

```
# real charging power sent to the battery
real_power_rl = []
for i in range(log_RL.__len__()):
    log_RL.loc[i, "hour_id"] = (log_RL.loc[i, "Time"].hour + log_RL.loc[i, "Time"].
↪minute / 60)

real_power_dumb = []
for i in range(dumb_log.__len__()):
    dumb_log.loc[i, "hour_id"] = (dumb_log.loc[i, "Time"].hour + dumb_log.loc[i, "Time"].
↪minute / 60)

# computing the average for each quarter hour over the entire year
mean_per_hid_rl = log_RL.groupby("hour_id").mean()["Charging energy"].reset_
↪index(drop=True)
mean_all_rl = []
for i in range(mean_per_hid_rl.__len__()):
    mean_all_rl.append(np.mean(mean_per_hid_rl[i]))

mean_per_hid_dumb = dumb_log.groupby("hour_id").mean()["Charging energy"].reset_
↪index(drop=True)
mean_all_dumb = []
for i in range(mean_per_hid_dumb.__len__()):
    mean_all_dumb.append(np.mean(mean_per_hid_dumb[i]))

# multiplied by the factor of 4 to go from kWh to kW (15 min time resolution)
mean_both = pd.DataFrame()
mean_both["RL"] = np.multiply(mean_all_rl, 4)
```

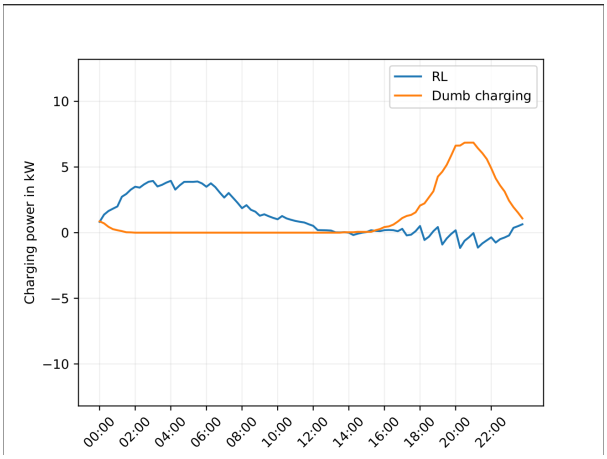
(continues on next page)

(continued from previous page)

```
mean_both["Dumb charging"] = np.multiply(mean_all_dumb, 4)

# plotting
mean_both.plot()
plt.xticks([0,8,16,24,32,40,48,56,64,72,80,88],
           ["00:00","02:00","04:00","06:00","08:00","10:00","12:00","14:00","16:00",
↪"18:00","20:00","22:00"],
           rotation=45)

plt.legend()
plt.grid(alpha=0.2)
plt.ylabel("Charging power in kW")
max = log_RL.loc[0, "Observation"][-10]
plt.ylim([-max * 1.2, max * 1.2])
plt.show()
```

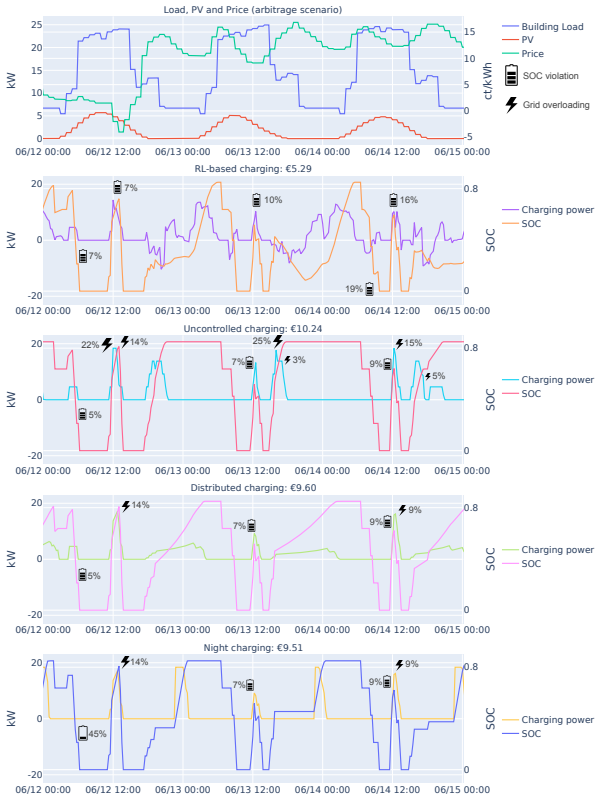


Saving the logs for future use

The logs can be saved as pickle files, so the same analytics and other visualizations can be performed on another machine, or at a later point in time.

```
dumb_log.to_pickle(f"dumb_log_{file_name_comment}")
log_rl.to_pickle(f"rl_log_{file_name_comment}")
```

Example of what can be done in post-processing



2.6 Benchmarking

Below, the logic of the three static benchmarks is presented, along with the linear optimization model. On github, fully implemented benchmarking notebooks are uploaded for each benchmark.

Uncontrolled charging

Every time step, the maximum charging signal is sent so that the battery is immediately charged upon arrival.

```
for i in range(episode_length * timesteps_per_hour * n_episodes):
    if dumb_norm_vec_env.env_method("is_done")[0]:
        dumb_norm_vec_env.reset()
    dumb_norm_vec_env.step([np.ones(n_evs)])
```

Distributed charging

In the distributed charging strategy, the charging process is spread out across the entire stay, and the battery only reaches full charge before arrival. The method `get_dist_factor` is used to get the laxity / distribution factor for each time step: time needed / time left.

```
for i in range(episode_length * timesteps_per_hour * n_episodes):
if dist_norm_vec_env.env_method("is_done")[0]:
    dist_norm_vec_env.reset()
dist_norm_vec_env.step(([np.clip(np.multiply(np.ones(n_evs), dist_norm_vec_env.env_
↪method("get_dist_factor")[0]),0,1)]))
```

Night charging

Night charging preferably starts charging at midnight, but begins earlier if the battery requires a longer charging time. The earliest departure time globally is considered to ensure a full battery in the worst case scenario. During the day between 11 and 14, a charging signal is sent to the battery to accommodate for the lunch break in the caretaker use-case.

```
df = env.db
df_leaving_home = df[(df['Location'].shift() == 'home') & (df['Location'] == 'driving')]
earliest_dep_time = df_leaving_home['date'].dt.time.min()
day_of_earliest_dep = df_leaving_home[df_leaving_home['date'].dt.time == earliest_dep_
↪time]['date'].min()
earliest_dep = earliest_dep_time.hour + earliest_dep_time.minute/60

evse = env.load_calculation.evse_max_power
cap = env.ev_conf.init_battery_cap
target_soc = env.ev_conf.target_soc
eff = env.ev_conf.charging_eff

max_time_needed = target_soc * cap / eff / evse  # time needed to charge to target soc␣
↪from 0
difference = earliest_dep - max_time_needed
starting_time = (24 + difference)
if starting_time > 24:
    starting_time = 23.99 # always start just before midnight

charging_hour = int(math.modf(starting_time)[1])
minutes = np.asarray([0, 15, 30, 45])
# split number and decimals, use decimals and choose the closest minute
closest_index = np.abs(minutes - int(math.modf(starting_time)[0]*60)).argmin()
charging_minute = minutes[closest_index]

episode_length = n_steps
n_episodes = n_episodes
night_norm_vec_env.reset()

charging=False

for i in range(episode_length * timesteps_per_hour * n_episodes):
    if night_norm_vec_env.env_method("is_done")[0]:
        night_norm_vec_env.reset()
    time: pd.Timestamp = night_norm_vec_env.env_method("get_time")[0]
    if ((time.hour >= 11) and (time.hour <= 14)) and (use_case=="ct"):
        night_norm_vec_env.step(([np.clip(np.multiply(np.ones(n_evs), night_norm_vec_env.
↪env_method("get_dist_factor")[0]),0,1)]))
        continue
    time: pd.Timestamp = night_norm_vec_env.env_method("get_time")[0]
    if (((charging_hour <= time.hour) and (charging_minute <= time.minute)) or␣
↪(charging)):
        if not charging:
            charging_start: pd.Timestamp = copy(time)
        charging=True
        night_norm_vec_env.step([np.ones(n_evs)])
    else:
        night_norm_vec_env.step([np.zeros(n_evs)])
    if charging and ((time - charging_start).total_seconds()/3600 > int(max_time_
```

(continues on next page)

(continued from previous page)

```
↪needed)):
        charging = False
```

Linear optimization model

The linear optimization model does 2 things: first, it calculates an optimal result, based on the available information of the FleetRL use-case (building load, vehicle schedules, etc.). Its objective is cost minimization and its variables are the actions in the range of [0,1] -> identical to the RL agents. Once a result has been obtained, it is run on a FleetRL environment -> the linear agent steps through the env and generates the log file that saves charging cost, SoH, violations, etc.

```
# importing necessary libraries
import datetime as dt
import numpy as np
import math
import matplotlib.pyplot as plt
from typing import Literal

from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3 import PPO, TD3
from stable_baselines3.common.vec_env import VecNormalize, SubprocVecEnv
from stable_baselines3.common.env_util import make_vec_env

from FleetRL.fleet_env.fleet_environment import FleetEnv

import pandas as pd
import pyomo.environ as pyo

# wrapper for parallelization
if __name__ == "__main__":

    # define parameters here for easier change
    n_steps = 8600
    n_episodes = 1
    n_evs = 5
    n_envs = 1
    time_steps_per_hour = 4
    use_case: str = "lmd"  # for file name
    scenario: Literal["arb", "real"] = "real"

    # environment arguments
    env_kwargs = {"schedule_name": "5_lmd_eval.csv",
                  "building_name": "load_lmd.csv",
                  "use_case": "lmd",
                  "include_building": True,
                  "include_pv": True,
                  "time_picker": "static",
                  "deg_emp": False,
                  "include_price": True,
                  "ignore_price_reward": False,
                  "ignore_invalid_penalty": True,
                  "ignore_overcharging_penalty": True,
                  "ignore_overloading_penalty": False,
```

(continues on next page)

(continued from previous page)

```
                "episode_length": n_steps,
                "normalize_in_env": False,
                "verbose": 0,
                "aux": True,
                "log_data": True,
                "calculate_degradation": True
                }

    # adapting price information according to user input
    if scenario == "real":
        env_kwargs["spot_markup"] = 10
        env_kwargs["spot_mul"] = 1.5
        env_kwargs["feed_in_ded"] = 0.25
        env_kwargs["price_name"] = "spot_2021_new_tariff.csv"
        env_kwargs["tariff_name"] = "fixed_feed_in.csv"
    elif scenario == "arb":
        env_kwargs["spot_markup"] = 0
        env_kwargs["spot_mul"] = 1
        env_kwargs["feed_in_ded"] = 0
        env_kwargs["price_name"] = "spot_2021_new.csv"
        env_kwargs["tariff_name"] = "spot_2021_new.csv"

    # creating vec_env for linear agent
    lin_vec_env = make_vec_env(FleetEnv,
                               n_envs=n_envs,
                               vec_env_cls=SubprocVecEnv,
                               env_kwargs=env_kwargs)

    # normalization
    lin_norm_vec_env = VecNormalize(venv=lin_vec_env,
                                    norm_obs=True,
                                    norm_reward=True,
                                    training=True,
                                    clip_reward=10.0)

    # creating an env object for accessing information in the pyomo model
    env = FleetEnv(use_case=use_case,
                   schedule_name=env_kwargs["schedule_name"],
                   tariff_name=env_kwargs["tariff_name"],
                   price_name=env_kwargs["price_name"],
                   episode_length=n_steps,
                   time_picker_name=env_kwargs["time_picker"],
                   building_name=env_kwargs["building_name"],
                   spot_markup=env_kwargs["spot_markup"],
                   spot_mul=env_kwargs["spot_mul"],
                   feed_in_ded=env_kwargs["feed_in_ded"])

    # reading the input file as a pandas DataFrame
    df: pd.DataFrame = env.db

    # Extracting information from the df
    ev_data = [df.loc[df["ID"]==i, "There"] for i in range(n_evs)]
```

(continues on next page)

(continued from previous page)

```
    building_data = df["load"]  # building load in kW

    # length of time, load, and pv series (with multiple EVs, this is the index)
    length_time_load_pv = 8760 * 4

    # price and tariff data
    price_data = np.multiply(np.add(df["DELU"], env.ev_conf.fixed_markup), env.ev_conf.
↪variable_multiplier) / 1000
    tariff_data = np.multiply(df["tariff"], 1-env.ev_conf.feed_in_deduction) / 1000

    # pv data
    pv_data = df["pv"]  # pv power in kW

    # soc on return, separately for each EV
    soc_on_return = [df.loc[df["ID"]==i, "SOC_on_return"] for i in range(n_evs)]

    # further model parameters
    battery_capacity = env.ev_conf.init_battery_cap  # EV batt size in kWh
    p_trafo = env.load_calculation.grid_connection  # Transformer rating in kW
    charging_eff = env.ev_conf.charging_eff  # charging losses
    discharging_eff = env.ev_conf.discharging_eff  # discharging losses
    init_soc = env.ev_conf.def_soc  # init SoC
    evse_max_power = env.load_calculation.evse_max_power  # kW, max rating of the charger

    # create pyomo model
    model = pyo.ConcreteModel(name="sc_pyomo")

    # create sets for the pyomo optimization
    model.timestep = pyo.Set(initialize=range(length_time_load_pv))
    model.time_batt = pyo.Set(initialize=range(0, length_time_load_pv+1))
    model.ev_id = pyo.Set(initialize=range(n_evs))

    # model parameters, i,j used in case multiple EVs exist
    model.building_load = pyo.Param(model.timestep, initialize={i: building_data[i] for␣
↪i in range(length_time_load_pv)})
    model.pv = pyo.Param(model.timestep, initialize={i: pv_data[i] for i in range(length_
↪time_load_pv)})
    model.ev_availability = pyo.Param(model.timestep, model.ev_id, initialize={(i, j):␣
↪ev_data[j].iloc[i] for i in range(length_time_load_pv) for j in range(n_evs)})
    model.soc_on_return = pyo.Param(model.timestep, model.ev_id, initialize={(i, j): soc_
↪on_return[j].iloc[i] for i in range(length_time_load_pv) for j in range(n_evs)})
    model.price = pyo.Param(model.timestep, initialize={i: price_data[i] for i in␣
↪range(length_time_load_pv)})
    model.tariff = pyo.Param(model.timestep, initialize={i: tariff_data[i] for i in␣
↪range(length_time_load_pv)})

    # decision variables
    # this assumes only charging, I could also make bidirectional later
    model.soc = pyo.Var(model.time_batt, model.ev_id, bounds=(0, env.ev_conf.target_soc))
    model.charging_signal = pyo.Var(model.timestep, model.ev_id, within=pyo.
↪NonNegativeReals, bounds=(0,1))
    model.discharging_signal = pyo.Var(model.timestep, model.ev_id, within=pyo.
```

(continues on next page)

(continued from previous page)

```
↪NonPositiveReals, bounds=(-1,0))
    model.positive_action = pyo.Var(model.timestep, model.ev_id, within=pyo.Binary)
    model.used_pv = pyo.Var(model.timestep, model.ev_id, within=pyo.NonNegativeReals)

    # constraints
    def grid_limit(m, i, ev):
        return ((m.charging_signal[i, ev] + m.discharging_signal[i, ev]) * evse_max_power
                + m.building_load[i] - m.pv[i] <= p_trafo)

    # charging an discharging cannot occur at the same time, big M method
    def mutual_exclusivity_charging(m, i, ev):
        return m.charging_signal[i, ev] <= m.positive_action[i, ev]

    def mutual_exclusivity_discharging(m, i, ev):
        return m.discharging_signal[i, ev] >= (m.positive_action[i, ev] - 1)

    # PV prioritized over grid
    def pv_use(m, i, ev):
        return m.used_pv[i, ev] <= m.charging_signal[i, ev] * evse_max_power

    # only use as much PV as available
    def pv_avail(m, i, ev):
        return m.used_pv[i, ev] <= m.pv[i] / n_evs

    def no_charge_when_no_car(m, i, ev):
        if m.ev_availability[i, ev] == 0:
            return m.charging_signal[i, ev] == 0
        else:
            return pyo.Constraint.Feasible

    def no_discharge_when_no_car(m, i, ev):
        if m.ev_availability[i, ev] == 0:
            return m.discharging_signal[i, ev] == 0
        else:
            return pyo.Constraint.Feasible

    def soc_rules(m, i, ev):
        #last time step
        if i == length_time_load_pv-1:
            return (m.soc[i+1, ev]
                    == m.soc[i, ev] + (m.charging_signal[i, ev]*charging_eff + m.
↪discharging_signal[i, ev])
                    * evse_max_power * 1 / time_steps_per_hour / battery_capacity)

        # new arrival
        elif (m.ev_availability[i, ev] == 0) and (m.ev_availability[i+1, ev] == 1):
            return m.soc[i+1, ev] == m.soc_on_return[i+1, ev]

        # departure in next time step
        elif (m.ev_availability[i, ev] == 1) and (m.ev_availability[i+1, ev] == 0):
            return m.soc[i, ev] == env.ev_conf.target_soc
```

(continues on next page)

(continued from previous page)

```
        else:
            return pyo.Constraint.Feasible

    def charging_dynamics(m, i, ev):
        #last time step
        if i == length_time_load_pv-1:
            return (m.soc[i+1, ev]
                    == m.soc[i, ev] + (m.charging_signal[i, ev]*charging_eff + m.
↪discharging_signal[i, ev])
                    * evse_max_power * 1 / time_steps_per_hour / battery_capacity)

        # charging
        if (m.ev_availability[i, ev] == 1) and (m.ev_availability[i+1, ev] == 1):
            return (m.soc[i+1, ev]
                    == m.soc[i, ev] + (m.charging_signal[i, ev]*charging_eff + m.
↪discharging_signal[i, ev])
                    * evse_max_power * 1 / time_steps_per_hour / battery_capacity)

        elif (m.ev_availability[i, ev] == 1) and (m.ev_availability[i+1, ev] == 0):
            return m.soc[i+1, ev] == 0

        elif m.ev_availability[i, ev] == 0:
            return m.soc[i, ev] == 0

        else:
            return pyo.Constraint.Feasible

    def max_charging_limit(m, i, ev):
        return m.charging_signal[i, ev]*evse_max_power <= evse_max_power * m.ev_
↪availability[i, ev]

    def max_discharging_limit(m, i, ev):
        return m.discharging_signal[i, ev]*evse_max_power*-1 <= evse_max_power * m.ev_
↪availability[i, ev]

    def first_soc(m, i, ev):
        return m.soc[0, ev] == init_soc

    def no_departure_abuse(m, i, ev):
        if i == length_time_load_pv - 1:
            return pyo.Constraint.Feasible
        if (m.ev_availability[i, ev] == 0) and (m.ev_availability[i-1, ev]) == 1:
            return m.discharging_signal[i, ev] == 0
        elif (m.ev_availability[i, ev] == 1) and (m.ev_availability[i+1, ev]) == 0:
            return m.discharging_signal[i, ev] == 0
        else:
            return pyo.Constraint.Feasible

    # constraints
    model.cs1 = pyo.Constraint(model.timestep, model.ev_id, rule=first_soc)
    model.cs2 = pyo.Constraint(model.timestep, model.ev_id, rule=grid_limit)
    model.cs3 = pyo.Constraint(model.timestep, model.ev_id, rule=max_charging_limit)
```

(continues on next page)

(continued from previous page)

```
    model.cs4 = pyo.Constraint(model.timestep, model.ev_id, rule=max_discharging_limit)
    model.cs5 = pyo.Constraint(model.timestep, model.ev_id, rule=soc_rules)
    model.cs6 = pyo.Constraint(model.timestep, model.ev_id, rule=charging_dynamics)
    model.cs8 = pyo.Constraint(model.timestep, model.ev_id, rule=mutual_exclusivity_
↪charging)
    model.cs9 = pyo.Constraint(model.timestep, model.ev_id, rule=mutual_exclusivity_
↪discharging)
    model.cs10 = pyo.Constraint(model.timestep, model.ev_id, rule=no_charge_when_no_car)
    model.cs11 = pyo.Constraint(model.timestep, model.ev_id, rule=no_discharge_when_no_
↪car)
    model.cs12 = pyo.Constraint(model.timestep, model.ev_id, rule=pv_use)
    model.cs13 = pyo.Constraint(model.timestep, model.ev_id, rule=pv_avail)

    timestep_set = pyo.RangeSet(0, length_time_load_pv-1)

    # cost minimization
    def obj_fun(m):
        return (sum([((m.charging_signal[i, ev] * evse_max_power - m.used_pv[i, ev]) /␣
↪time_steps_per_hour) * m.price[i] +
                     ((m.discharging_signal[i, ev] * evse_max_power * discharging_eff) /␣
↪time_steps_per_hour) * m.tariff[i]
                     for i in m.timestep for ev in range(n_evs)]))

    # change solver here to glpk if gurobi not configured
    model.obj = pyo.Objective(rule=obj_fun, sense=pyo.minimize)
    opt = pyo.SolverFactory('gurobi')#, executable="/home/enzo/Downloads/gurobi10.0.2_
↪linux64/gurobi1002/linux64/")
    # for quicker solving
    opt.options['mipgap'] = 0.005
    # print additional information
    res = opt.solve(model, tee=True)
    print(res)

    # extract actions array for each time step, this is the result of the optimization
    actions = [np.array([model.charging_signal[i,j].value + model.discharging_signal[i,
↪j].value for j in range(n_evs)]) for i in range(length_time_load_pv)]
    actions = pd.DataFrame({"action": actions})

    # set the same index as for the RL agent and the other benchmarks
    actions.index = pd.date_range(start="2020-01-01 00:00", end="2020-12-30 23:59", freq=
↪"15T")
    actions["hid"] = actions.index.hour + actions.index.minute/60

    # plot the resulting action curve from the pyomo optimization
    len_day = 24*4
    action_plot = []
    for i in range(len_day):
        action_plot.append(actions.groupby("hid").mean()["action"].reset_
↪index(drop=True)[i].mean())
    plt.plot(action_plot)
    plt.show()
```

(continues on next page)

(continued from previous page)

```
    # feed the resulting actions into the FleetRL environment to get log data and KPIs
    lin_norm_vec_env.reset()
    start_time = lin_norm_vec_env.env_method("get_start_time")[0]
    end_time = pd.to_datetime(start_time) + dt.timedelta(hours=n_steps)
    env_actions = actions.loc[(actions.index >= start_time) & (actions.index <= end_
↪time), "action"].reset_index(drop=True)

    for i in range(n_steps*time_steps_per_hour):
        lin_norm_vec_env.step([np.multiply(np.ones(n_evs), env_actions[i])])

    # get log from environment
    lin_log: pd.DataFrame = lin_norm_vec_env.env_method("get_log")[0]

    # plotting analog to RL evaluation
    lin_log.reset_index(drop=True, inplace=True)
    lin_log = lin_log.iloc[0:-2]

    real_power_lin = []
    for i in range(lin_log.__len__()):
        lin_log.loc[i, "hour_id"] = (lin_log.loc[i, "Time"].hour + lin_log.loc[i, "Time
↪"].minute / 60)

    mean_per_hid_lin = lin_log.groupby("hour_id").mean()["Charging energy"].reset_
↪index(drop=True)
    mean_all_lin = []
    for i in range(mean_per_hid_lin.__len__()):
        mean_all_lin.append(np.mean(mean_per_hid_lin[i]))

    mean = pd.DataFrame()
    mean["Distributed charging"] = np.multiply(mean_all_lin, 4)

    mean.plot()

    plt.xticks([0,8,16,24,32,40,48,56,64,72,80,88]
               ,["00:00","02:00","04:00","06:00","08:00","10:00","12:00","14:00","16:00",
↪"18:00","20:00","22:00"],
               rotation=45)

    plt.legend()
    plt.grid(alpha=0.2)

    plt.ylabel("Charging power in kW")
    max = lin_log.loc[0, "Observation"][-10]
    plt.ylim([-max * 1.2, max * 1.2])

    plt.show()

    # save log as pickle
    lin_log.to_pickle("lin_log.pickle")
```

2.7 Fleet environment

2.7.1 Subpackages

fleetrl.fleet_env.config package

Submodules

fleetrl.fleet_env.config.ev_config module

class fleetrl.fleet_env.config.ev_config.**EvConfig**(*env_config*)

Bases: object

- The EV config sets some default parameters regarding the vehicles, charging infrastructure, efficiencies, etc.

fleetrl.fleet_env.config.score_config module

class fleetrl.fleet_env.config.score_config.**ScoreConfig**(*env_config*)

Bases: object

The Score Config sets coefficients to calculate the reward function - Multipliers: price_multiplier, penalty_invalid_action, penalty_overcharging, penalty_overloading - Changing the multipliers to 0 leads to ignoring this aspect of the reward function - The SOC violation and overloading are calculated using sigmoid functions

overloading_penalty(*rel_loading*)

static soc_violation_penalty(*missing_soc*)

fleetrl.fleet_env.config.time_config module

class fleetrl.fleet_env.config.time_config.**TimeConfig**(*env_config*)

Bases: object

The Time config sets parameters for the episode and MDP dynamics. - end_cutoff: Default at 60. The last two months are not regarding in the training observations - The last two months are used as a validation set during training

2.7.2 Submodules

2.7.3 Episode module

class fleetrl.fleet_env.episode.**Episode**(*time_conf*)

Bases: object

The Episode class holds all / most of the values that are episode-specific

2.7.4 Environment module

class fleetrl.fleet_env.fleet_environment.**FleetEnv**(*env_config*)

Bases: Env

FleetRL: Reinforcement Learning environment for commercial vehicle fleets. Author: Enzo Alexander Cording - https://github.com/EnzoCording Master's thesis project, M.Sc. Sustainable Energy Engineering @ KTH Copyright (c) 2023, Enzo Cording

This framework is built on the gymnasium core API and inherits from it. __init__, reset, and step are implemented, calling other modules and functions where needed. Base-derived class architecture is implemented, and the code is structured in a modular manner to enable improvements or changes in the model.

Only publicly available data or own-generated data has been used in this implementation.

The agent only sees information coming from the chargers: SOC, how long the vehicle is still plugged in, etc. However, this framework matches the number of chargers with the number of cars to reduce complexity. If more cars than chargers should be modelled, an allocation algorithm is necessary. What is more, battery degradation is modelled in this environment. In this case, the information of the car is required (instead of the charger). Modelling is facilitated by matching cars and chargers one-to-one. Therefore, throughout the code, "car" and "ev_charger" might be used interchangeably as indices.

Note that this does not present a simplification from the agent perspective because the agent does only handles the SOC and time left at the charger, regardless of whether the vehicle is matching the charger one-to-one or not.

adjust_caretaker_lunch_soc()

The caretaker target SOC can be set lower during the lunch break to avoid unfair penalties occurring. This is because the break is not long enough to charge until 0.85 target SOC. :return: None -> sets the target SOC during lunch break hours to 0.65 by default

adjust_score_config()

auto_gen()

This function automatically generates schedules as specified. Uses the ScheduleGenerator module. Note: this can take up to 1-3 hours, depending on the number of vehicles.

Returns

None -> The schedule is generated and placed in the input folder

change_markups()

check_data_paths(*input_path*, *schedule_path*, *spot_path*, *load_path*, *pv_path*)

choose_observer()

This function chooses the right observer, depending on whether to include price, building, PV, etc. :return: obs (Observer) -> The observer module to choose

choose_time_picker(*time_picker*)

Chooses the right time picker based on the specified in input string. Static: Always the same time is picked to start an episode Random: Start an episode randomly from the training set Eval: Start an episode randomly from the validation set :type time_picker: :param time_picker: (string), specifies which time picker to choose: "static", "eval", "random" :return: tp (TimePicker) -> time picker object

close()

After the user has finished using the environment, close contains the code necessary to "clean up" the environment.

This is critical for closing rendering windows, database or HTTP connections.

detect_dim_and_bounds()

This function chooses the right dimension of the observation space based on the chosen configuration. Each increase of dim is explained below. The low_obs and high_obs are built in the normalizer object, using the dim value that was calculated in this function.

- set boundaries of the observation space, detects if normalized or not.
- If aux flag is true, additional information enlarges the observation space.
- The following code goes through all possible environment setups.
- Depending on the setup, the dimensions differ and every case is handled differently.

Returns

low_obs and high_obs: tuple[float, float] | tuple[np.ndarray, np.ndarray] -> used for gym.Spaces

get_dist_factor()

This function returns the distribution/laxity factor: how much time needed vs. how much time left at charger If factor is 0.1, the dist agent would only charge with 10% of the EVSE capacity. Call via env_method("get_dist_factor")[0] if using an SB3 Vectorized Environment :return: dist/laxity factor, float

get_log()

This function can be called through SB3 vectorized environments via VecEnv.env_method("get_log")[0] The zero index is required so only the first element -> the DataFrame is extracted

Returns

Log dataframe

get_next_dt()

Calculates the time delta from the current time step and the next one. This allows for csv input files that have irregular time intervals. Energy calculations will automatically adjust for the dynamic time differences through kWh = kW * dt

Returns

next time delta in hours

get_next_minutes()

Calculates the integer of minutes until the next time step. This therefore limits the framework's current maximum resolution to discrete time steps of 1 min. This will be improved soon, as well as the dependency to know the future value beforehand.

Returns

Integer of minutes until next timestep

get_start_time()

VecEnv.env_method("get_start_time")[0] :return: pd.TimeStamp

get_time()

VecEnv.env_method("get_time")[0] :return: pd.TimeStamp: current timestamp

is_done()

VecEnv.env_method("is_done")[0] :return: Flag is episode is done, bool

print(*action*)

The print function can provide useful information of the environment dynamics and the agent's actions. Can slow down FPS due to the printing at each timestep

Parameters

action – Action of the agent

Returns
None -> Just prints information if specified

`read_config`(*conf_path*)

`render`()

Compute the render frames as specified by `render_mode` during the initialization of the environment.

The environment's `metadata` render modes (*env.metadata["render_modes"]*) should contain the possible ways to implement the render modes. In addition, list versions for most render modes is achieved through *gymnasium.make* which automatically applies a wrapper to collect rendered frames.

Note:
As the `render_mode` is known during `__init__`, the objects used to render the environment state should be initialised in `__init__`.

By convention, if the `render_mode` is:

- None (default): no render is computed.
- "human": The environment is continuously rendered in the current display or terminal, usually for human consumption. This rendering should occur during *step()* and *render()* doesn't need to be called. Returns `None`.
- "rgb_array": Return a single frame representing the current state of the environment. A frame is a `np.ndarray` with shape `(x, y, 3)` representing RGB values for an x-by-y pixel image.
- "ansi": Return a strings (`str`) or `StringIO.StringIO` containing a terminal-style text representation for each time step. The text can include newlines and ANSI escape sequences (e.g. for colors).
- "rgb_array_list" and "ansi_list": List based version of render modes are possible (except Human) through the wrapper, `gymnasium.wrappers.RenderCollection` that is automatically applied during `gymnasium.make(..., render_mode="rgb_array_list")`. The frames collected are popped after *render()* is called or *reset()*.

Note:
Make sure that your class's `metadata` `"render_modes"` key includes the list of supported modes.

Changed in version 0.25.0: The render function was changed to no longer accept parameters, rather these parameters should be specified in the environment initialised, i.e., `gymnasium.make("CartPole-v1", render_mode="human")`

`reset`(***kwargs*)

Parameters
kwargs – Necessary for gym inheritance

Return type
`tuple[array, dict]`

Returns
First observation (either normalized or not) and an info dict

`set_start_time`(*start_time*)

VecEnv.env_method("set_start_time", [f"{start_time}"]) Must parse the function and argument of start_time :type start_time: `str` :param start_time: string of pd.TimeStamp / date :return: None

`specify_company_and_battery_size`(*use_case*)

step(*actions*)

The main logic of the EV charging problem is orchestrated in the step function. Input: Action on charging power for each EV Output: Next state, reward

Intermediate processes: EV charging model, battery degradation, cost calculation, building load, penalties, etc.

The step function runs as long as the done flag is False. Different functions and modules are called in this function to reduce the complexity and to distribute the tasks of the model.

Parameters
actions (array) – Actions parsed by the agent, from -1 to 1, representing % of kW of the EVSE

Return type
tuple[array, float, bool, bool, dict]

Returns
Tuple containing next observation, reward, done, truncated and info dictionary

2.8 Benchmarking

2.8.1 Submodules

2.8.2 Benchmarking base class

class fleetrl.benchmarking.benchmark.**Benchmark**

Bases: object

Parent class for benchmark modules.

plot_benchmark(*log*)

Parameters
log (DataFrame) – Log dataframe

Return type
None

Returns
None, plots the benchmark

run_benchmark(*use_case*, *env_kwargs*, *seed=None*)

This method contains the logic of the respective benchmarks, executes it on the given environment and returns a log.

Parameters

- **use_case** (str) – String that specifies use-case (“lmd”, “ct”, “ut”)
- **env_kwargs** (dict) – Environment parameters
- **seed** (int) – seed for RNG

Return type
DataFrame

Returns
Log Dataframe of the benchmark, can be saved as pickle

2.8.3 Distributed charging

class fleetrl.benchmarking.distributed_charging.**DistributedCharging**(*n_steps*, *n_evs*, *n_episodes=1*, *n_envs=1*, *time_steps_per_hour=4*)

Bases: *Benchmark*

plot_benchmark(*dist_log*)

Parameters
log – Log dataframe

Return type
None

Returns
None, plots the benchmark

run_benchmark(*use_case*, *env_kwargs*, *seed=None*)

This method contains the logic of the respective benchmarks, executes it on the given environment and returns a log.

Parameters

- **use_case** (str) – String that specifies use-case (“lmd”, “ct”, “ut”)
- **env_kwargs** (dict) – Environment parameters
- **seed** (int) – seed for RNG

Return type
DataFrame

Returns
Log Dataframe of the benchmark, can be saved as pickle

2.8.4 Linear optimisation

class fleetrl.benchmarking.linear_optimization.**LinearOptimization**(*n_steps*, *n_evs*, *n_episodes=1*, *n_envs=1*, *time_steps_per_hour=4*)

Bases: *Benchmark*

plot_benchmark(*lin_log*)

Parameters
log – Log dataframe

Return type
None

Returns
None, plots the benchmark

run_benchmark(*use_case*, *env_kwargs*, *seed=None*)

This method contains the logic of the respective benchmarks, executes it on the given environment and returns a log.

Parameters

- **use_case** (str) – String that specifies use-case (“lmd”, “ct”, “ut”)
- **env_kwargs** (dict) – Environment parameters
- **seed** (int) – seed for RNG

Return type
DataFrame

Returns
Log Dataframe of the benchmark, can be saved as pickle

2.8.5 Night charging

class fleetrl.benchmarking.night_charging.**NightCharging**(*n_steps*, *n_evs*, *n_episodes=1*, *n_envs=1*, *time_steps_per_hour=4*)

Bases: *Benchmark*

plot_benchmark(*night_log*)

Parameters
log – Log dataframe

Return type
None

Returns
None, plots the benchmark

run_benchmark(*use_case*, *env_kwargs*, *seed=None*)

This method contains the logic of the respective benchmarks, executes it on the given environment and returns a log.

Parameters

- **use_case** (str) – String that specifies use-case (“lmd”, “ct”, “ut”)
- **env_kwargs** (dict) – Environment parameters
- **seed** (int) – seed for RNG

Return type
DataFrame

Returns
Log Dataframe of the benchmark, can be saved as pickle

2.8.6 Uncontrolled charging

class fleetrl.benchmarking.uncontrolled_charging.**Uncontrolled**(*n_steps*, *n_evs*, *n_episodes=1*, *n_envs=1*, *time_steps_per_hour=4*)

Bases: *Benchmark*

plot_benchmark(*dumb_log*)

Parameters
log – Log dataframe

Return type
None

Returns
None, plots the benchmark

run_benchmark(*use_case*, *env_kwargs*, *seed=None*)

This method contains the logic of the respective benchmarks, executes it on the given environment and returns a log.

Parameters

- **use_case** (str) – String that specifies use-case ("lmd", "ct", "ut")
- **env_kwargs** (dict) – Environment parameters
- **seed** (int) – seed for RNG

Return type
DataFrame

Returns
Log Dataframe of the benchmark, can be saved as pickle

2.9 Agent evaluation

2.9.1 Submodules

2.9.2 Evaluation base class

class fleetrl.agent_eval.evaluation.**Evaluation**

Bases: object

compare(*rl_log*, *benchmark_log*)

evaluate_agent(*env_kwargs*, *norm_stats_path*, *model_path*, *seed=None*)

plot_action_dist(*rl_log*, *benchmark_log*)

plot_detailed_actions(*start_date*, *end_date*, *rl_log=None*, *uc_log=None*, *dist_log=None*, *night_log=None*, *lp_log=None*)

plot_soh(*rl_log*, *benchmark_log*)

plot_violations(*rl_log*, *benchmark_log*)

2.9.3 Basic evaluation

class fleetrl.agent_eval.basic_evaluation.**BasicEvaluation**(*n_steps*, *n_evs*, *n_envs=1*, *n_episodes=1*)

Bases: *Evaluation*

compare(*rl_log*, *benchmark_log*)

evaluate_agent(*env_kwargs*, *norm_stats_path*, *model_path*, *seed=None*)

plot_action_dist(*rl_log*, *benchmark_log*)

plot_detailed_actions(*start_date*, *end_date*, *rl_log=None*, *uc_log=None*, *dist_log=None*, *night_log=None*, *lp_log=None*)

plot_soh(*rl_log*, *benchmark_log*)

plot_violations(*rl_log*, *benchmark_log*)

2.10 Data logger

2.10.1 Submodules

2.10.2 Data logger

class fleetrl.utils.data_logger.data_logger.**DataLogger**(*episode_length*)

Bases: object

Logs data to allow for postprocessing, graphs, etc. The log is a dataframe, where each row can be a float or an array. Deepcopy to avoid risk of mutability (logs pointing back to changing variables)

log_data(*time*, *obs*, *action*, *reward*, *cashflow*, *penalty*, *grid*, *soc_v*, *degradation*, *charge_log*, *soh*)

While the env object is the same, the episode counter will recognise different episodes. A dict is created with the required values, and then appended to the log dataframe

Parameters

- **time** (Timestamp) – Current timestamp
- **obs** (ndarray) – Observation array
- **action** (ndarray) – Action array
- **reward** (float) – Reward float
- **cashflow** (float) – in EUR
- **penalty** (float) – penalty float
- **grid** (float) – Grid connection in kW
- **soc_v** (float) – Amount of SOC violated in # of batteries
- **degradation** (float) – Degradation in that timestep
- **charge_log** (ndarray) – How much energy flowed into the batteries in kWh
- **soh** (ndarray) – Current SoH, array

Returns

None

2.11 Load calculation

2.11.1 Submodules

2.11.2 Load calculation

class fleetrl.utils.load_calculation.load_calculation.**CompanyType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Caretaker = 2

Custom = 4

Delivery = 1

Utility = 3

class fleetrl.utils.load_calculation.load_calculation.**LoadCalculation**(*env_config*, *company_type*, *max_load*, *num_cars*)

Bases: object

The Load Calculation class sets grid connection, and calculates load overloading

check_violation(*actions*, *there*, *building_load*, *pv*)

Parameters

- **actions** (list[float]) – Actions list, action for each EV [-1,1]
- **there** (list[int]) – Flag is EV is plugged in or not [0;1]
- **building_load** (float) – Current building load in kW
- **pv** (float) – Current PV in kW

Return type
(bool, float)

Returns

2.12 Normalisation

2.12.1 Submodules

2.12.2 Normalisation base class

class fleetrl.utils.normalization.normalization.**Normalization**

Bases: object

Parent class for Normalization.

make_boundaries(*dim*)

Create the observation space for gym. Depending on the observation, upper and lower boundaries can change.

Parameters
dim (tuple[int]) – The length of the array, depends on the number of cars, and other information contained in the obs

Return type
tuple[float, float] | tuple[ndarray, ndarray]

Returns
A tuple, containing the low obs and high obs array that will be parsed to gym.Spaces.box

normalize_obs(*obs*)

Normalizes the values in an observation.

Parameters
obs (dict) – An observation, containing the state of charge, hours left, and price

Return type
ndarray

Returns
The normalized observation, concatenated to a single array

2.12.3 Oracle normalisation

class fleetrl.utils.normalization.oracle_normalization.**OracleNormalization**(*db*, *building_flag*, *pv_flag*, *price_flag*, *ev_conf*, *load_calc*, *aux*)

Bases: *Normalization*

Oracle Normalization assumes the knowledge of the max and min values of the dataset. This is necessary to perform a global min/max normalization. Alternatively, a rolling normalization could be implemented.

static flatten_obs(*obs*)

Observations must be flattened for openAI gym compatibility. The parsed observation must be a flat array and not a dictionary. The dictionary either includes float or array. The function removes the nesting.

Parameters
obs – Normalized observation dictionary

Returns
A flattened array - necessary for the RL algorithms to be in a 1-dim array e.g. [v_1, . . . , v_N]

make_boundaries(*dim*)

The boundaries are 0 and 1 because the observations are min/max normalized.

Parameters
dim (tuple[int]) – Dimension of the observation depending on the flags

Return type
tuple[float, float] | tuple[ndarray, ndarray]

Returns
Low and high observation arrays for gym.Spaces.

normalize_obs(*input_obs*)

Normalization function. Different cases are checked depending on the flags of PV, load, price, and aux. Input observations are a dictionary with clear namings, to make further changes in the code easy and readable.

Parameters
input_obs (dict) – Input observation: Un-normalized observations as specified in the observer.

Return type
ndarray

Returns
Normalized observation.

2.12.4 Unit normalisation

class fleetrl.utils.normalization.unit_normalization.**UnitNormalization**

Bases: *Normalization*

This function does not normalize, but parses the un-normalized values.

static flatten_obs(*obs*)

Flattening nested dict to np.ndarray

Parameters
obs – Observation dictionary including floats or arrays

Returns
Flattened observation

make_boundaries(*dim*)

The bounds can be -inf and inf. This is the least restricting path and allows for cross-compatibility of agents on environments that have the same shape.

Parameters
dim (tuple[int]) – Dimension depending on the flags.

Return type
tuple[float, float] | tuple[ndarray, ndarray]

Returns
Low_obs and high_obs for gym.Spaces.

normalize_obs(*obs*)

Obs is flattened and returned. :type obs: dict :param obs: Observation dictionary that includes floats or array :rtype: ndarray :return: Flattened np.ndarray

2.13 Observation

2.13.1 Submodules

2.13.2 Observer base class

class fleetrl.utils.observation.observer.**Observer**

Bases: object

Parent class for observer modules.

get_obs(*db*, *price_lookahead*, *bl_pv_lookahead*, *time*, *ev_conf*, *load_calc*, *aux*, *target_soc*)

Parameters

- **db** (DataFrame) – database from the env
- **price_lookahead** (int) – lookahead window for spot price
- **bl_pv_lookahead** (int) – lookahead window for building load and pv
- **time** (Timestamp) – current time of time step
- **ev_conf** (*EvConfig*) – EV config needed for batt capacity and other params
- **load_calc** (*LoadCalculation*) – Load calc module needed for grid connection and other params
- **aux** (bool) – Include auxiliary information that might help the agent to learn the problem
- **target_soc** (list) – A list of target soc values, one for each car

Return type
dict

Returns
Returns a list of np arrays that make up different parts of the observation.

static get_trip_len(*db*, *car*, *time*)

Parameters

- **db** (DataFrame) – from the env
- **car** (int) – car ID
- **time** (Timestamp) – current timestamp

Return type
float

Returns
length of trip in hours as a float

2.13.3 Building load and PV

class fleetrl.utils.observation.observer_bl_pv.**ObserverWithBoth**

Bases: *Observer*

Observer for price, PV, and load (full model).

get_obs(*db*, *price_lookahead*, *bl_pv_lookahead*, *time*, *ev_conf*, *load_calc*, *aux*, *target_soc*)

- define the starting and ending time via lookahead, np.where returns the index in the dataframe
- add lookahead + 2 here because of rounding issues with the resample function on square times (00:00)
- get data of price and date from the specific indices
- resample data to only include one value per hour (the others are duplicates)
- only take into account the current value, and the specified hours of lookahead

Parameters

- **db** (DataFrame) – Database from env
- **price_lookahead** (int) – Lookahead in hours for price
- **bl_pv_lookahead** (int) – Lookahead in hours for PV and building
- **time** (Timestamp) – Current time
- **ev_conf** (*EvConfig*) – EV config data, used for battery capacity, etc.
- **load_calc** (*LoadCalculation*) – Load calc module, used for grid connection, etc.
- **aux** (bool) – Flag to include extra information on the problem or not. Can help with training
- **target_soc** (list) – List for the current target SOC of each car

Return type
dict

Returns
Dict of lists with different parts of the observation

static get_trip_len(*db*, *car*, *time*)

Parameters

- **db** (DataFrame) – from the env
- **car** (int) – car ID
- **time** (Timestamp) – current timestamp

Return type
float

Returns
length of trip in hours as a float

2.13.4 Price only

class fleetrl.utils.observation.observer_price_only.**ObserverPriceOnly**

Bases: *Observer*

Observer for a model that only takes into account the price, but disregards PV and building load.

get_obs(*db*, *price_lookahead*, *bl_pv_lookahead*, *time*, *ev_conf*, *load_calc*, *aux*, *target_soc*)

- define the starting and ending time via lookahead, np.where returns the index in the dataframe
- add lookahead + 2 here because of rounding issues with the resample function on square times (00:00)
- get data of price and date from the specific indices
- resample data to only include one value per hour (the others are duplicates)
- only take into account the current value, and the specified hours of lookahead

Parameters

- **db** (DataFrame) – Database from env
- **price_lookahead** (int) – Lookahead in hours for price
- **bl_pv_lookahead** (int) – Lookahead in hours for PV and building
- **time** (Timestamp) – Current time
- **ev_conf** (*EvConfig*) – EV config data, used for battery capacity, etc.
- **load_calc** (*LoadCalculation*) – Load calc module, used for grid connection, etc.
- **aux** (bool) – Flag to include extra information on the problem or not. Can help with training
- **target_soc** (list) – List for the current target SOC of each car

Return type
dict

Returns
Dict of lists with different parts of the observation

static get_trip_len(*db*, *car*, *time*)

Parameters

- **db** (DataFrame) – from the env
- **car** (int) – car ID
- **time** (Timestamp) – current timestamp

Return type
float

Returns
length of trip in hours as a float

2.13.5 SOC and time only

class fleetrl.utils.observation.observer_soc_time_only.**ObserverSocTimeOnly**

Bases: *Observer*

Observer that only regards SOC and time left, but not charging cost, PV or building laod

get_obs(*db*, *price_lookahead*, *bl_pv_lookahead*, *time*, *ev_conf* , *load_calc*, *aux*, *target_soc*)

- define the starting and ending time via lookahead, np.where returns the index in the dataframe
- add lookahead + 2 here because of rounding issues with the resample function on square times (00:00)
- get data of price and date from the specific indices
- resample data to only include one value per hour (the others are duplicates)
- only take into account the current value, and the specified hours of lookahead

Parameters

- **db** (DataFrame) – Database from env
- **price_lookahead** (int) – Lookahead in hours for price
- **bl_pv_lookahead** (int) – Lookahead in hours for PV and building
- **time** (Timestamp) – Current time
- **ev_conf** (*EvConfig*) – EV config data, used for battery capacity, etc.
- **load_calc** (*LoadCalculation*) – Load calc module, used for grid connection, etc.
- **aux** (bool) – Flag to include extra information on the problem or not. Can help with training
- **target_soc** (list) – List for the current target SOC of each car

Return type
dict

Returns
Dict of lists with different parts of the observation

static get_trip_len(*db*, *car*, *time*)

Parameters

- **db** (DataFrame) – from the env
- **car** (int) – car ID
- **time** (Timestamp) – current timestamp

Return type
float

Returns
length of trip in hours as a float

2.13.6 Building load

class fleetrl.utils.observation.observer_with_building_load.**ObserverWithBuildingLoad**

Bases: *Observer*

Observer that takes into account price, and building load.

get_obs(*db*, *price_lookahead*, *bl_pv_lookahead*, *time*, *ev_conf* , *load_calc*, *aux*, *target_soc*)

- define the starting and ending time via lookahead, np.where returns the index in the dataframe
- add lookahead + 2 here because of rounding issues with the resample function on square times (00:00)
- get data of price and date from the specific indices
- resample data to only include one value per hour (the others are duplicates)
- only take into account the current value, and the specified hours of lookahead

Parameters

- **db** (DataFrame) – Database from env
- **price_lookahead** (int) – Lookahead in hours for price
- **bl_pv_lookahead** (int) – Lookahead in hours for PV and building
- **time** (Timestamp) – Current time
- **ev_conf** (*EvConfig*) – EV config data, used for battery capacity, etc.
- **load_calc** (*LoadCalculation*) – Load calc module, used for grid connection, etc.
- **aux** (bool) – Flag to include extra information on the problem or not. Can help with training
- **target_soc** (list) – List for the current target SOC of each car

Return type
dict

Returns
Dict of lists with different parts of the observation

static get_trip_len(*db*, *car*, *time*)

Parameters

- **db** (DataFrame) – from the env
- **car** (int) – car ID
- **time** (Timestamp) – current timestamp

Return type
float

Returns
length of trip in hours as a float

2.13.7 PV

class fleetrl.utils.observation.observer_with_pv.**ObserverWithPV**

Bases: *Observer*

Observer that regards price, and PV.

get_obs(*db*, *price_lookahead*, *bl_pv_lookahead*, *time*, *ev_conf*, *load_calc*, *aux*, *target_soc*)

- define the starting and ending time via lookahead, np.where returns the index in the dataframe
- add lookahead + 2 here because of rounding issues with the resample function on square times (00:00)
- get data of price and date from the specific indices
- resample data to only include one value per hour (the others are duplicates)
- only take into account the current value, and the specified hours of lookahead

Parameters

- **db** (DataFrame) – Database from env
- **price_lookahead** (int) – Lookahead in hours for price
- **bl_pv_lookahead** (int) – Lookahead in hours for PV and building
- **time** (Timestamp) – Current time
- **ev_conf** (*EvConfig*) – EV config data, used for battery capacity, etc.
- **load_calc** (*LoadCalculation*) – Load calc module, used for grid connection, etc.
- **aux** (bool) – Flag to include extra information on the problem or not. Can help with training
- **target_soc** (list) – List for the current target SOC of each car

Return type
dict

Returns
Dict of lists with different parts of the observation

static get_trip_len(*db*, *car*, *time*)

Parameters

- **db** (DataFrame) – from the env
- **car** (int) – car ID
- **time** (Timestamp) – current timestamp

Return type
float

Returns
length of trip in hours as a float

2.14 Schedule

2.14.1 Submodules

2.14.2 Schedule config

class fleetrl.utils.schedule.schedule_config.**ScheduleConfig**(*schedule_type*, *env_config*)

Bases: object

Statistical configurations for the schedule generator. Mean and standard deviation values are specified for each metric, allowing for a distributional and probabilistic generation approach.

class fleetrl.utils.schedule.schedule_config.**ScheduleType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Caretaker = 2

Custom = 4

Delivery = 1

Utility = 3

2.14.3 Schedule generator

class fleetrl.utils.schedule.schedule_generator.**ScheduleGenerator**(*env_config*, *schedule_type=ScheduleType.Delivery*, *vehicle_id='0'*)

Bases: object

Probabilistic schedule generator. Loops through each 15 min timeslot in the yearly dataframe and generates a row entry. The format is kept similar to emobpy to enable compatability and ease of use.

generate_caretaker()

Caretaker generator. Lunch break, operations on Sunday, chance for emergency trips at night :return: pd.DataFrame of the schedule

generate_custom()

Custom schedule generator. Saturdays operations occur but at reduced levels, no operations on Sunday. :return: pd.DataFrame of the schedule

generate_delivery()

Delivery schedule generator. Saturdays operations occur but at reduced levels, no operations on Sunday. :return: pd.DataFrame of the schedule

generate_schedule()

This method chooses the right generation method depending on the use-case. Returns the schedule dataframe.

Returns
pd.DataFrame of the schedule

generate_utility()

Utility generation. Chance for operations on Sunday. :return: pd.DataFrame of the schedule.

2.15 Time picker

2.15.1 Submodules

2.15.2 Time picker base class

class fleetrl.utils.time_picker.time_picker.**TimePicker**

Bases: object

choose_time(*db*, *freq*, *end_cutoff*)

Parent class for time picker objects :type db: Series :param db: dataframe from env :type freq: str :param freq: frequency specification string for pandas :type end_cutoff: int :param end_cutoff: amount of days cut off at the end to allow some buffer. In the eval time picker case, the end cutoff specifies the size of the validation set. :rtype: Timestamp :return: A chosen time stamp

2.15.3 Eval time picker

class fleetrl.utils.time_picker.eval_time_picker.**EvalTimePicker**(*ep_len*)

Bases: *TimePicker*

Time picker for validation set. The amount of days and thus the train/validation split is set in time config.

choose_time(*db*, *freq*, *end_cutoff*)

Randomly chooses a start time from the validation set.

Parameters

- **db** (Series) – Database
- **freq** (str) – Time frequency
- **end_cutoff** (int) – This is the size of the validation window. By default, the end_cutoff is 2 months, so Nov and Dec are the validation set.

Return type
Timestamp

Returns
start time, pd.TimeStamp

2.15.4 Random time picker

class fleetrl.utils.time_picker.random_time_picker.**RandomTimePicker**

Bases: *TimePicker*

Picks a random time from the training set.

choose_time(*db*, *freq*, *end_cutoff*)

Randomly chooses a start time from the validation set.

Parameters

- **db** (Series) – Database
- **freq** (str) – Time frequency
- **end_cutoff** (int) – Buffer that avoids problem with lookaheads

Return type
`Timestamp`

Returns
start time, pd.TimeStamp

2.15.5 Static time picker

class fleetrl.utils.time_picker.static_time_picker.**StaticTimePicker**(*start_time='01/01/2021 00:00'*)

Bases: *TimePicker*

Picks a static / always the same starting time.

choose_time(*db*, *freq*, *end_cutoff*)

Parent class for time picker objects :type db: `Series` :param db: dataframe from env :type freq: `str` :param freq: frequency specification string for pandas :type end_cutoff: `int` :param end_cutoff: amount of days cut off at the end to allow some buffer. In the eval time picker case, the end cutoff specifies the size of the validation set. :rtype: `Timestamp` :return: A chosen time stamp

2.16 Data processing

2.16.1 Submodules

2.16.2 Data processing

class fleetrl.utils.data_processing.data_processing.**DataLoader**(*path_name*, *schedule_name*, *spot_name*, *tariff_name*, *building_name*, *pv_name*, *time_conf*, *ev_conf*, *target_soc*, *building_flag*, *pv_flag*, *real_time*)

Bases: `object`

The DataLoader class handles the csv import and pre-processing of the timeseries information. Optimized pandas functions such as merge_asof are used to significantly speed up processing compared to loops. Cython could further improve this initial processing step. It only happens once when instantiating env objects.

compute_from_schedule(*ev_conf*, *time_conf*, *target_soc*)

This function pre-processes the input data and adds additional rows to the file. There flag, time left at charger, soc on return, consumption, etc.

Use merge_asof and vectorized operations for performance gains

Returns
None

load_building_load(*path_name*, *file_name*, *date_range*)

Load building load from csv :type path_name: :param path_name: Parent directory string :type file_name: :param file_name: file name with .csv ending :type date_range: :param date_range: pd.date_range which was defined from the "date" column in the EV schedules. Note that the EV schedule dates therefore dictate the model's timeframe. :return: load dataframe

load_feed_in(*path_name*, *tariff_name*, *date_range*)

Load feed-in from csv :type path_name: :param path_name: Parent directory string :type tariff_name: :param tariff_name: file name with .csv ending :type date_range: :param date_range: pd.date_range which was defined from the "date" column in the EV schedules. Note that the EV schedule dates therefore dictate the model's timeframe. :return: tariff dataframe

load_prices(*path_name*, *spot_name*, *date_range*)

Load prices from csv :type path_name: :param path_name: Parent directory string :type spot_name: :param spot_name: file name with .csv ending :type date_range: :param date_range: pd.date_range which was defined from the "date" column in the EV schedules. Note that the EV schedule dates therefore dictate the model's timeframe. :return: spot price dataframe

load_prices_original(*path_name*, *spot_name*, *date_range*)

Load prices from csv :type path_name: :param path_name: Parent directory string :type spot_name: :param spot_name: file name with .csv ending :type date_range: :param date_range: pd.date_range which was defined from the "date" column in the EV schedules. Note that the EV schedule dates therefore dictate the model's timeframe. :return: spot price dataframe

load_pv(*path_name*, *pv_name*, *date_range*)

Load pv from csv :type path_name: :param path_name: Parent directory string :type pv_name: :param pv_name: file name with .csv ending :type date_range: :param date_range: pd.date_range which was defined from the "date" column in the EV schedules. Note that the EV schedule dates therefore dictate the model's timeframe. :return: pv dataframe

static shape_price_reward(*db*, *ev_conf*)

- de-trend prices, so they can be used as a reward function
- agent should not be penalised more if the general price level is higher
- instead, the agent should just focus on price fluctuations and exploit them
- computing average for whole year, split data into monthly chunks
- offset monthly chunks, such that the monthly average = annual average
- this corrects for absolute price increases, but leaves fluctuations intact

Parameters

- **db** (DataFrame) – database with schedules, pv, prices, load, dataframe
- **ev_conf** (*EvConfig*) – ev config object

Returns

db with updated, de-trended prices

2.17 EV charging

2.17.1 Submodules

2.17.2 EV charger

class fleetrl.utils.ev_charging.ev_charger.**EvCharger**(*ev_conf*)

Bases: object

The EV Charger class handles the logic from action to battery charging. SOC and charging cost are calculated. Two different cost scenarios can be modelled: spot market in both directions (allowing for arbitrage) and a commercial electricity tariff that takes into account grid fees, and markups. Discharging at PV feed-in levels.

charge(*db*, *num_cars*, *actions*, *episode*, *load_calculation*, *ev_conf*, *time_conf*, *score_conf*, *print_updates*, *target_soc*)

The function loops through each car separately and computes SOC and charging cost depending on the action. Positive actions -> charging, negative actions -> discharging. Penalties are taken into account if the battery would be overcharged (agent sends a charging action to a full battery).

Parameters

- **db** (DataFrame) – The schedule database of the EVs
- **num_cars** (int) – Number of cars in the model
- **actions** – Actions taken by the agent
- **episode** (*Episode*) – Episode object with its parameters and functions
- **load_calculation** (*LoadCalculation*) – Load calc object with its parameters and functions
- **ev_conf** (*EvConfig*) – Config of the EVs
- **time_conf** (*TimeConfig*) – Time configuration
- **score_conf** (*ScoreConfig*) – Score and penalty configuration
- **print_updates** (bool) – Bool whether to print statements or not (maybe lower fps)
- **target_soc** (list) – target soc for each car

Returns

soc, next soc, the reward and the monetary value (cashflow)

2.18 Battery degradation

2.18.1 Submodules

2.18.2 Degradation base class

class fleetrl.utils.battery_degradation.batt_deg.**BatteryDegradation**

Bases: object

calculate_degradation(*soc_list*, *charging_power*, *time_conf*, *temp*)

This is the parent class for degradation methods. Any new implemented method must follow this style of inputs and outputs. Then, the method can be used in FleetRL by changing one line of code in the import.

The degradation methods in FleetRL are implemented such that degradation is calculated in real-time. In the step method of the environment class, the current SoH is calculated by SoH -= degradation

Parameters

- **soc_list** (list) – Historic values of SoC until now
- **charging_power** (float) – Charging power in kW

- **time_conf** (*TimeConfig*) – Time config instance, necessary for time step length
- **temp** (float) – Temperature to use in battery degradation in °C

Return type
float

Returns
Degradation of battery (unit-less, reduction of SoH, which is max. 1)

2.18.3 Method comparison

class fleetrl.utils.battery_degradation.compare_methods.**Comparison**

Bases: object

compare_methods(*data*, *save=False*)

deg_rate_calendar(*t*, *avg_soc*, *temp*)

deg_rate_cycle(*dod*, *avg_soc*, *temp*)

emp_deg(*data*)

l_with_sei(*fd*)

static l_without_sei(*self*, *l*, *fd*)

rainflow_sei(*soc_log*)

stress_dod(*dod*)

stress_soc(*soc*)

stress_temp(*temp*)

stress_time(*t*)

2.18.4 Empirical degradation

class fleetrl.utils.battery_degradation.empirical_degradation.**EmpiricalDegradation**(*init_soh*, *num_cars*)

Bases: *BatteryDegradation*

calculate_degradation(*soc_log*, *charging_power*, *time_conf*, *temp*)

Similar to non-linear SEI, the most recent event is taken, and the linear-based degradation is calculated. No rainflow counting, thus degradation is computed for each time step.

- find out the most recent entries in the soc list
- get old and new soc
- get average soc
- compute cycle and calendar based on avg soc and charging power

Parameters

- **soc_log** (list) – Historical log of SOC

- **charging_power** (float) – EVSE power in kW
- **time_conf** (*TimeConfig*) – time config object
- **temp** (float) – temperature

Return type
array

Returns
Degradation, float

2.18.5 Degradation logger

class fleetrl.utils.battery_degradation.log_data_deg.**LogDataDeg**(*episode*)

Bases: object

Data Logger for degradation purposes. Deepcopy to avoid mutability

add_log_entry()

log_soc(*soc*)

log_soh(*soh*)

2.18.6 Non-linear degradation (Rainflow SEI)

class fleetrl.utils.battery_degradation.rainflow_sei_degradation.**RainflowSeiDegradation**(*init_soh*, *num_cars*)

Bases: *BatteryDegradation*

Non-linear battery degradation model, Xu et al. - Source: Modeling of Lithium-Ion Battery Degradation for Cell Life Assessment - https://ieeexplore.ieee.org/document/7488267

calculate_degradation(*soc_log*, *charging_power*, *time_conf*, *temp*)

Calculates degradation. SOC from environment is converted to the necessary format for rainflow counting. Every time step, this function is called with a new soc_log entry. The rainflow library calculates the resulting cycle counts, which do not grow identically with soc_log entries. Rainflow results are used for the SOH calculations. If a new rainflow result entry appears, a calculation iteration is conducted. For calculation, the second most recent rainflow result is used.

Parameters

- **soc_log** (list) – SOC list for each time step t: t1:[soc_car1, soc_car2, ...], t2:[soc_car1, soc_car2,...]
- **charging_power** (float) – kW of the charger
- **time_conf** (*TimeConfig*) – Time config instance from the environment
- **temp** (float) – Temperature (default at 25°C)

Return type
array

Returns
Numpy array of degradation for each vehicle [deg_1, deg_2, ...]

deg_rate_calendar(*t*, *avg_soc*, *temp*)

deg_rate_cycle(*dod*, *avg_soc*, *temp*)

l_with_sei(*fd*)

static l_without_sei(*l*, *fd*)

stress_dod(*dod*)

stress_soc(*soc*)

stress_temp(*temp*)

stress_time(*t*)

2.19 Event manager

2.19.1 Submodules

2.19.2 Event manager

class fleetrl.utils.event_manager.event_manager.**EventManager**

Bases: object

The EventManager class decides if a new event occurred that is relevant for the RL agent For example: Subsequent steps where no new car arrives are flagged as irrelevant and not sent to the agent New arrival triggers an observation to be sent to the RL agent via the step() return function

Idea from: https://doi.org/10.1016/j.buildenv.2023.110546

static check_event(*episode*)

Parameters
events – variable that contains number of relevant events

Return type
bool

Returns
bool to confirm that a relevant event took place

2.20 Rendering

2.20.1 Submodules

2.20.2 Render

class fleetrl.utils.rendering.render.**ParkingLotRenderer**

Bases: object

`static render`(*there*, *kw*, *soc*)

This function provides a basic graphical overview of the current charging situation. Currently done via Matplotlib but will be expanded to pygame soon.

Parameters

- **there** – Vector with booleans if spot is taken or not
- **kw** – Vector with amount of kW in each spot
- **soc** – Vector with SOC in each spot

Returns

None - plots a graphical representation of the current charging situation

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

E

F

G

I

L

M

N

O

P

R