Report

The purpose of the work is to create a framework capable of extracting real world data in terms of both traffic demand and traffic layout of various points on the map of a city, then optimize the traffic lights in real time around an area to minimize the total waiting time of the cars, or any other metrics that are relevant to traffic optimization. The methods used behind are based on Deep Reinforcement Learning, and the end user can extend the framework from various points to provide different optimization targets (rewards), observations (what information is available from the traffic system), algorithms for traffic lights agents, etc.

The overall idea is that we have two modes:

1. Single-agent – A single RL agent controls all the traffic lights in the given area
2. Multi-agent – There is an individual agent per traffic light, acting cooperatively as much as possible as a Zero-Sum game.

Github address: <https://github.com/unibuc-cs/TrafficFlowOptimization>

# 1. Datasets

1. Public data benchmarks

We are using RESCO dataset for evaluation and experiments,

Paper: <https://people.engr.tamu.edu/guni/Papers/NeurIPS-signals.pdf>

Github: <https://github.com/LucasAlegre/sumo-rl>

<https://github.com/LucasAlegre/sumo-rl/blob/master/nets/RESCO/maps.png>

Check folder nets/RESCO. It contains real data with different complexities:

Diagram

Description automatically generated

1. Bucharest data and optimization for a real case

* Using OSMWizard and SUMO we can capture real data from Bucharest. Check folder MASSA\_Bucuresti for a few examples(see the other document in this folder to see how I did this).
* Problem: we don’t have real traffic data. We tried to take from cameras, it is possible but only 2-3 intersections cameras are working …
* How to real data:
  + O/D matrices into SUMO

<https://sumo.dlr.de/docs/Demand/Importing_O/D_Matrices.html#describing_the_taz>

* + Import Induction loop data <https://sumo.dlr.de/docs/TraCI/Induction_Loop_Value_Retrieval.html>
  + Bolt data (as promised): We can map the traffic O/D, traffic demand on lanes and intersections.

1. See the other research doc in this folder for other methods to get real kind of data from Paris and other locations in the world.

# 2. Current RL method implemented

## A. MDP definition

**Observation**

The default observation for each traffic signal agent is a vector:

**obs = [phase\_one\_hot, min\_green, lane\_1\_density,...,lane\_n\_density, lane\_1\_queue,...,lane\_n\_queue]**

**phase\_one\_hot** : one-hot encoded vector indicating the current active green phase

**min\_green** is a binary variable indicating whether min\_green seconds have already passed in the current phase

**lane\_i\_density** is the number of vehicles in incoming lane i dividided by the total capacity of the lane

**lane\_i\_queue** the number of queued (speed below 0.1 m/s) vehicles in incoming lane i divided by the total capacity of the lane

We can define own observation changing the method 'compute\_observation' of TrafficSignal.

**Actions**

The action space is discrete. Every 'delta\_time' seconds, each traffic signal agent can choose the next green phase configuration. Every time a phase change occurs, the next phase is preeceded by a yellow phase lasting yellow\_time seconds. E.g.: In the 2-way single intersection there are |A| = 4 discrete actions, corresponding to the following green phase configurations:

A screenshot of a computer

Description automatically generated with low confidence

**Rewards**

How much the total delay (sum of the waiting times of all approaching vehicles) changed in relation to the previous time-step.

A picture containing text, clock

Description automatically generated

We can customize reward function within method 'compute\_reward' of TrafficSignal.

## B. Software stack

A picture containing diagram

Description automatically generated

* Single agent means that there will be a single agent controlling all the traffic lights in the environment/ In this case we use Baselines3 to leverage existing RL core implementation. In this moment we use DQN only with a simple MLP to map states to a latent.
* Multi-agent means that there will be an individual agent assigned to each traffic light. In this case, we use RLLib for massive parallelization on clusters, processors, and GPUs.

This case also needs pre-processing operations such as normalizing the length of actions/observations, etc. We use *supersuit* package for this. As algorithm, now A3C is used.

## C. Experiments and Code discussion

C1. Usage

The code for both single agent uses Baselines3 and can be found in experiments/massa\_rl\_baselines3\_dpq.py. The code for multi-agent methods can be found in massa\_marl\_a3c.py. Both derive some common utilities from massa\_common.py. You can find for all these experiments the attached PyCharm configuration so if you install it, it is easy to understand the arguments and running setup.

To output the plots and get mean basic results according to a series of run, use plot.py. As an explanation of the output of this script, below are some result samples from training the agents during a series of episodes. The X axis shows the second in the simulator (mapped to the one used in the dataset), while the Y axis shows the total waiting time of all stopped cars that wait at a stop light. Ideally, this should be as low as possible. The graphs show the average waiting time (blue curve line), and deviation (between minimum and maximum) at each point in time.

* + Single agent (DQN):

Chart

Description automatically generated

* + Multi agent (A3C):
  + Chart, histogram

    Description automatically generated

Hint: name the experiment using parameter *-outputbasepath* ( as show in the PyCharm configuration examples) to facilitate the evaluation of results.

C2. Environment and communication between it and the SUMMO simulator

The base environment class SumoEnvironment handles the communication with SUMMO. Check the SumoEnvironment ::\_\_init\_\_ function where we import the SUMMO Python API, find the binary (search for *sumolib.checkBinary* call) and execute it using the internal web-sockets protocol. The traffic lights are gathered inside SumoEnvironment ::*traffic\_signals* variable automatically and an *agent* is started for each in the case of multi-agent RL method. Otherwise, if single-agent is used, an agent controlling all traffic lights is used.

The SumoEnvironment ::*start\_simulation* function calls the starting and sets the parameters of the simulation (e.g. start / end time, etc).

**Class TrafficSignal** (located in traffic\_signal.py) is the one representing an agent logic. What it does is:

* Represent a Traffic Signal of an intersection, and an agent in the case of multi-agent usage
* Performs the ***action*** – i.e., setting the next phase. which green phase is going to be open for the next (delta\_time) seconds. Handles the yellow between as well.
* Handles the observation, i.e. how each lane linked to itself looks like
* Can compute common statistics used by literature such as: lanes’ density, total queued vehicles, waiting time on average, waiting time per lane, pressure, etc.

All other macro-level statistics should be implemented here ideally.

* Based on the statistics it can compute *rewards* for the agent*.* Check function TrafficSignal ::compute\_reward
* The ***observation*** construction is handled by TrafficSignal::compute\_observation

**Important performance hint**: if you define LIBSUMO\_AS\_TRACI in the environment variables, then SUMMO will be used as a library only for simulation. You don’t have access to the GUI / rendering the simulation, but it will help evaluate the algorithms at ~3-4 times faster on average.

C3. Architecture

In the case of a **single-agent** specification, SummoEnvironment is enough. This can be easily wrapped around the environments in common RL libraries such as TFAgents or Baselines3, as example massa\_rl\_baseline3\_dqn.py shows.

**The full architecture** in the case multi-agent specification is depicted below:

Diagram

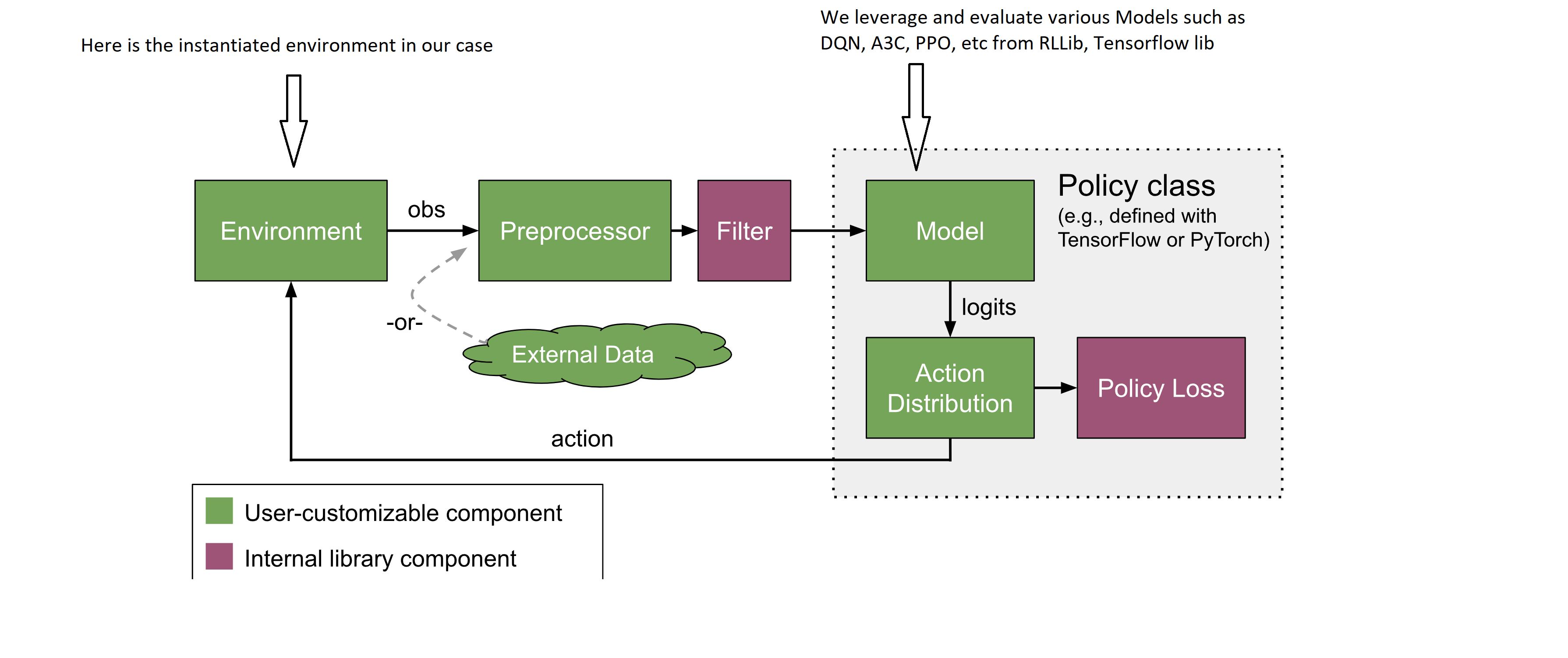
Description automatically generated

The purpose of SumoEnvironmentPZ class is to be the adapter between the AECEnv defined in the PeetinZoo library and the SumoEnviornment class.

The Wrappers layers are various classes that add (through inheritance) support for transforming the SumoEnviornment’s operations for pre/post processing of observations, actions, inputs, rewards. For example OutOfBoundaryWrapper is one that does checking of boundaries of the environment, while OrderEnforcerWrapper could enforce the action execution in a specific way. The Pad wrappers are used to make sure that observations and actions have the same shape (padding with 0s in general). The idea behind is that each agent could have a different number of lanes, thus the MDP definition given in C1 will have different observation spaces and actions by default.

The difference between *PeetingZooEnv* and *ParallelPeetingZooEnv,* is that the first considers that each agent knows what other agents decision was on each step, thus acts in a sequential fixed order. The second one uses the concept of Partially Observable Stochastic Games (POSGs)  , where the agent can’t see what other agents did on the current step, i.e. they act in parallel. From our point of view, this second model is the most realistic one and it is selected by default in the implementation.

To leverage **RLLib** infrastructure (<https://docs.ray.io/en/latest/rllib.html>) for distribution of learning and/or inference, the instantiated EnviornmnetInstace (one of the too PeetingZooEnv or ParallelPeetingZooEnv), is injected in the RLLib framework as in the following picture:



The action distribution comes back to *TrafficSignal* in the end, which acts according to the sampled action from Action Distribution’s result.

C3. Results comparisons

TODO: add results on 4-5 different algorithms to compare on a map.

# 3. Future work and publishing

A. Beat state of the art multi-agent using different observations, rewards, deep architectures (see the other doc).

B. Use data from Bolt in Bucharest to publish how car passengers can collaborate together to improve real time traffic signals automatically, with an IoT based infrastructure in the city. Also Create a reusable tool to automatically extract map’s locations from OSM, link to a datasource and produce datasets.

C. Compare and review state of the art work and algorithms on real data from RESCO, Bucharest, simulations.

D. Niche apps: optimize traffic using RESCO/Bolt by: smart pedestrian crossing, cut-off restrictions for bicycle lanes or bus temporarily, speed restrictions, etc.

Some ideas: <https://flow-project.github.io/publications.html>