WOLF

uoft

Aug 31, 2020

CONTENTS

WOLF (What to do to Optimise Large Flows).

CHAPTER

ONE

INTRODUCTION

WOLF (What to do to Optimise Large Flows) is a python API developed by the University of Toronto, in collaboration with Huawei Canada. Its aim is to offer Traffic Light Control optimisation on micro (Aimsum, SUMO) and macro (Cellular Transmission Model) simulators. It relies on Berkley's Flow API for traffic micro simulation (a wrapper around Aimsim and SUMO), and the Ray framework for multiprocessing and RL optimisation.

The main contribution of WOLF is a wrapper around Flow RL environment. Our wrapper allow a lot of customisation and more tool for traffic light optimisation (Flow lacks of those features). This wrapper is very similar to a gym environment. You can learn a single-agent or multi-agents policy to interact with this environment. WOLF uses RLLib (a sub-API of Ray) to optimise those policies, but you are free to use something else (openAI baselines, stable-baseline, dopamine, TF-agent etc), however you will lack some features we offer (visualisation scripts and plotting results). See *this section* for a functional use of our API.

In order to understand most of the code and configuration of WOLF, one needs to understand how Flow and Ray works.

CHAPTER

TWO

DESIGN PHILOSOPHY

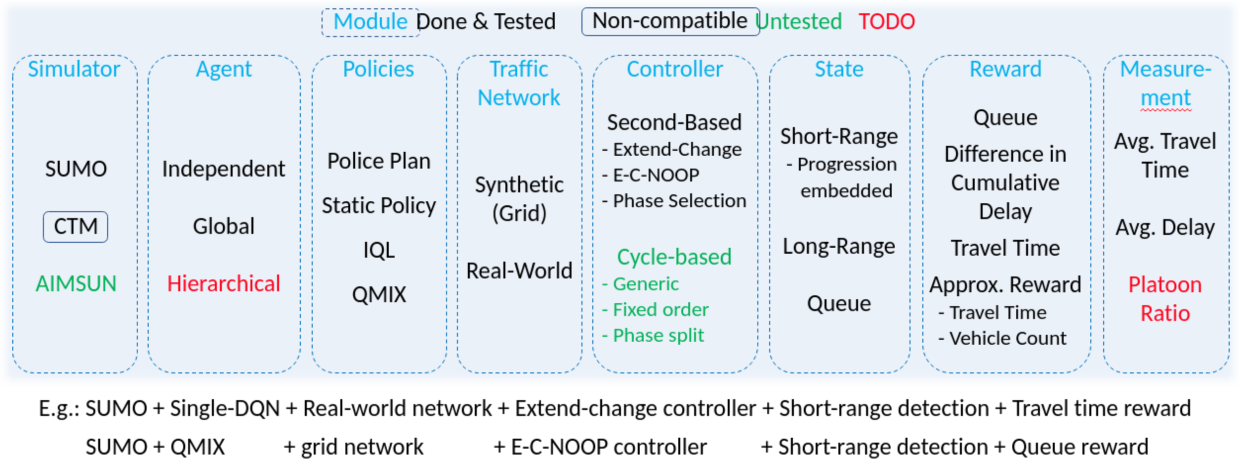
2.1 Composition Design Pattern

Everything is customisable: metrics, algorithms, controller, observation, simulator. To operate this, the architecture is designed following a composition pattern (instead of Hierarchical).

TrafficEnv is a gym/rllib/flow environment that combines different components. Most of the components can be found under wolf/world/environments/agents/connectors. But also in agent_factory.

In env_factories.py, you have a collection of prefabricated environments. You can choose the env you want by specifying “”env”: “test2” under the “config” key of your Ray experiment. The env key must exist in the registry (wolf/utils/configuration/registery.py)

Not that CTMEnv is unchangeable (or very lightly).



2.2 One Configuration File to Rule Them All

All the parameters can be specified in a configuration file (yaml or json). This file followed the same structure as any Ray configuration file. This is under “env_config” that you can compose your TrafficEnv (choosing the simulator, the action space etc). Like in Ray, you can add a “grid_search” entry as a parent of any key, in order to operate different simulations. A list must be under gridsearch. For example:

```
“a”:
    “gridsearch”:
        - “hello”: “world”
        - “hello2”: “world2”
```

This config means the parameter “a” will be instantiate two times, one with {“hello”: “world”} and the other time with {“hello2”: “world2”}.

The test configuration files work with ray (see the key “ray” in the config file), if you decide to use another RL library, you will need to decide your own config file structure, but you can keep everything under “general” and reuse everything under “env_config” (which are agnostic to the RL framework).

CHAPTER

THREE

THIRD PARTY FRAMEWORKS

3.1 Ray

Ray is an python API used for multiprocessing (Ray), meta-parameter search (Tune) and reinforcement learning (RLLib).

They offer a collection of single and multi-agents RL algorithms:

Algorithms and their parameters: https://docs.ray.io/en/latest/rllib-algorithms.html

We only tried PPO, DQN, APEX-DQN, QMIX and APEX-QMIX with the uoft framework, but others should work.

Make sure to understand how Ray work before diving into WOLF documentation.

3.2 Gym

Gym is a collection of Reinforcement Learning environments. These environments implement the famous Gym interface, where an agent interact with it through a function called "step". Agent receive observation, reward, information and a done flag after each step. Gym focuses on single agent formulation. Ray provides a multi-agents formulation that Flow will extend.

3.3 Flow

Flow is a wrapper around two micro simulators, Aimsum and SUMO. It comes with a gym-like/ray-like environment for Reinforcement Learning. The environment can simulate multiple agents in either Aimsum or SUMO. The API make is easier to create network programatically (like the GridEnv network), or load open street maps and simulate "real" networks. Primarily focus of flow is autonomous driving and the support for traffic light control is very limited. This is why provide a wrapper around Flow environment. Unlike Flow, our environment support all kind of components like controllers, rewards and observations, all compatible with a single container class, TrafficEnv.

CHAPTER

FOUR

INSTALLATION

To get wolf running, you need few things: wolf, SUMO (a traffic simulator), flow (a traffic simulation framework), and ray (a framework for building distributed ML applications) together with RLlib (a reinforcement learning library). In this documentation, we will lead you going through all the steps to make every component ready.

Once each component is installed successfully, you might get some missing module bugs from Python. Just install the missing module using your OS-specific package manager / installation tool. Follow the shell commands below to get started.

Wolf is designed to work with multiple traffic simulators: SUMO, AIMSUN, CTM (built-in). Wolf + SUMO is fully tested and ready to use now. Wolf is compatible with AIMSUN, but not fully tested yet.

4.1 Installing SUMO

There are two ways to install SUMO. You can either install the latest released version of SUMO by using `sudo apt-get`, or locally install SUMO by compiling from the source code. For more details, you can checkout the SUMO's documentation: https://sumo.dlr.de/docs/Downloads.php.

Add SUMO to your system

If you are free to use the `sudo` command, we recommand to install SUMO with the first method, which is easier. You can add the most recent SUMO to your Ubuntu system by doing these:

```
sudo add-apt-repository ppa:sumo/stable
sudo apt-get update
sudo apt-get install sumo sumo-tools sumo-doc
```

Then you have to add SUMO to your system path. Please add the following two lines into your `.bashrc` file.

```
export SUMO_HOME=/usr/share/sumo
export PATH=/usr/share/sumo/bin:$PATH
```

And source the `.bashrc` file.

```
source ~/.bashrc
```

Compile SUMO from source code

The other option for installing SUMO without `sudo` is that compile SUMO from the source code locally. We recommand you to install version 1.6.0 of SUMO which is fully tested and works well.

```
# download the code, unzip it, and put the folder to the place you want.
curl -O https://sumo.dlr.de/releases/1.6.0/sumo-src-1.6.0.tar.gz
tar xzvf sumo-src-1.6.0.tar.gz
```

(continues on next page)

(continued from previous page)

```
mv -r sumo-1.6.0 /the/path/to/sumo

# compile the code
cd /the/path/to/sumo
cmake .
make
```

You still need to add SUMO to your system path following the previous introduction. Please add the following two lines into your `.bashrc` file.

```
export SUMO_HOME=/the/path/to/sumo
export PATH=/the/path/to/sumo/bin:$PATH
```

And source the `.bashrc` file.

```
source ~/.bashrc
```

Finally, you can test your SUMO installation is successful or not with the following commands:

```
which sumo    # gives you the path
sumo    # shows you the version of SUMO
sumo-gui    # pop-up SUMO gui window
```

4.2 Installing flow

The original flow is a vehicle control oriented framework, which does not exactly match our tasks. But flow still has good and useful high-level simulation wrappers. Hence, we made lots of enhancements to let it fit our tasks. Here we'll introduce how to install the modified flow.

TODO: *Rephrase this paragraph*: Before installing all the following packages, please make a blank virtual environment with any tool (conda/vertualenv/docker).

```
# clone the source code from the github:
git clone https://github.com/RaptorMai/flow.git

# install the flow
cd flow
pip install -e .
```

4.3 Installing ray and RLlib

Ray project is evolving very fast. There is a dilemma in choosing the version of ray: the latest released version cannot meet all our demands, and the latest snapshots cannot be tested frequently and guaranteed to work. Hence, we pick a certain commit of ray which is fully tested and works well. Please follow the instruction below to install the correct version of ray.

```
# install the specific commit of ray
pip install https://ray-wheels.s3-us-west-2.amazonaws.com/master/
↪b62ec7787fd472b7f32dad7c18152706ba8c035e/ray-0.9.0.dev0-cp36-cp36m-manylinux1_x86_
↪64.whl
```

(continues on next page)

(continued from previous page)

```
# install ray-rllib
pip install ray[rllib]
```

4.4 Installing wolf

Finally, we will install wolf and finish the whole installation process. There are also two methods to let the wolf work: install wolf with `pip install`, or add wolf to the `PYTHONPATH`. With either method, you have to clone the source code first:

```
git clone http://116.66.187.35:4502/gitlab/its/sow45_code.git
```

Install wolf with pip

```
cd sow45_code
pip install -e .
```

Use wolf by specifying the PYTHONPATH

Please add the following line into your `.bashrc` file.

```
export PYTHONPATH=$PYTHONPATH:/path/to/sow45_code
```

And source the `.bashrc` file.

```
source ~/.bashrc
```

At last, please install the following extra packages:

```
pip install pandas
pip install matplotlib
pip install tensorflow_gpu
pip install torch
pip install imutils
```

4.5 Installing QMIX Dependencies

Installing QMIX Dependencies: sacred, tensorboard-logger

```
cd ..
git clone https://github.com/oxwhirl/sacred.git
cd sacred
sed -i '36s/.*/        tf.random.set_seed(seed)/' randomness.py
pip install -e .

pip install tensorboard-logger
```

CHAPTER FIVE

SCRIPTS

All script are under wolf/scripts.

5.1 The main script

main.py is the core script. It runs a configuration file with Ray as RL backend.

```
python main.py configs/main.yaml
```

We provide some tests under wolf/tests. Tests come as configuration file, so there is nothing to do beside calling main.py with the path of the test file, for example

```
python main.py ../tests/traffic_env/test0/global_agent.yaml
```

See the *configuration file* example for more details.

5.2 Master plot

If you did multiple experiments, and you want to plot everything on the same image. You can call the following script.

```
python master_plot.py --config example_plot.yaml
```

In the yaml file, you just need to specify the experiment folders and the color of each curve.

You can also choose what RLlib metric you want to compare under the metrics key. Metrics are the one displayed on tensorboard.

For example:

```
"max_x": 25 # the x axis upper bound
"experiments":
  "global_agent":
    "path_results": "/home/ncarrara/ray_results/global_agent_night_run" # path of the␣
↪experiment folder.
    "color": "blue"
  "random":
    "path_results": "/home/ncarrara/ray_results/random"
    "color": "red"
"metrics":
  "eps-greedy":
    "path": ["episode_reward_mean"] # Ray's metric "path", meaning the key in the␣
↪result file produced by Ray.
```

(continues on next page)

(continued from previous page)

```
  "greedy":
    "path": ["evaluation", "episode_reward_mean"] # This one mean evaluation/episode_
↪reward_mean.
```

5.3 Merge experiments

You can create multiples configs files with just one experiment by file (Ray's experiment). For example, dqn.yaml, and qmix.yaml:

```
"ray":
  "run_experiments":
    "experiments":
      "dqn":
        ...
```

```
"ray":
  "run_experiments":
    "experiments":
      "qmix":
        ...
```

That way you can run those experiments individually. If you want to run all those experiments at the same time, you can call the following script:

```
python merge_and_run_experiments.py --files path/to/file.yaml path/to/file2.yaml --
↪sumo_home /home/user/sumo_binaries/bin --workspace tmp
```

It will merge the config files into a single one:

```
"ray":
  "run_experiments":
    "experiments":
      "dqn":
        ...
      "qmix":
        ...
```

And ray will run each of those single experiment (sequentially or in parallel). This is particularly useful if you want to debug experiments individually, and compare all of them with a single run (without having to copy paste everything).

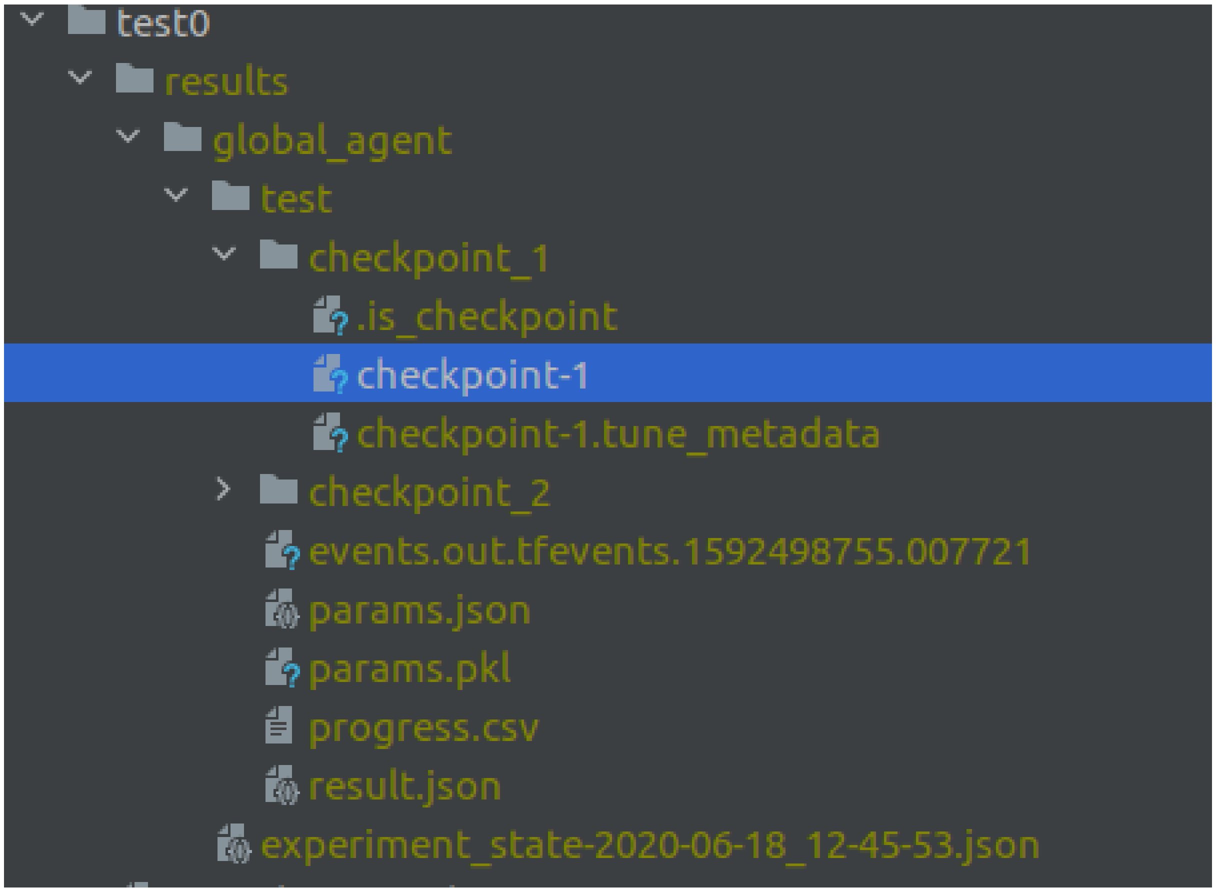
5.4 Visualisation

When your policy is being optimised, Ray will save on the disk the model. You can visualise the greedy policy using this script:

```
python rollout.py --checkpoint /where/the/checkpoint/is/save --run APEX --env␣
↪"traffic_env_test2" --video_dir /path/to/save/video --no_render
```

This script is just an extension of the rollout script from RLLib, tailored to use our custom environments (TrafficEnv and CTMEnv)

Usually the checkpoint is saved under the name of the experiment, under your workspace (specified in the config file), for example:



```
test0
  results
    global_agent
      test
        checkpoint_1
          .is_checkpoint
          checkpoint-1
          checkpoint-1.tune_metadata
        checkpoint_2
        events.out.tfevents.1592498755.007721
        params.json
        params.pkl
        progress.csv
        result.json
      experiment_state-2020-06-18_12-45-53.json
```

In that case, the checkpoint path should be:

```
tests/traffic_env/test0/results/global_agent/test/checkpoint_1/checkpoint-1
```

You must also make sure the "–run" arguments match the algorithm you used with RLLib to learn the policy.

The argument "–env" describe the entry of environment in the registry. It must correspond to the environment you use for learning.

Cf the registry for environment keys:

```
R.register_env_factory("simple_grid", simple_grid)
R.register_env_factory("generic_grid", generic_grid)
R.register_env_factory("grid_master_slaves_3", lambda config: grid_master_
↪slaves(config, 3, 300))
R.register_env_factory("grid_gaussian_master_slaves_4", lambda config: grid_gaussian_
↪master_slaves(config, 4, 300))
R.register_env_factory("traffic_env_test0", test0)
R.register_env_factory("traffic_env_test0_1", test0_1)
R.register_env_factory("traffic_env_test1", test1)
R.register_env_factory("traffic_env_test2", lambda config: grid_master_slaves(config,␣
↪4, 300))
R.register_env_factory("real_world_network", real_world_network)
R.register_env_factory("default_ctm", lambda config: CtmEnv.create_env(CtmEnv,
↪**config))
```

(continues on next page)

(continued from previous page)

```
R.register_env_factory("ctm_test1", lambda config: ctm_test1(config))
R.register_env_factory("ctm_test2", lambda config: ctm_test2(config))
R.register_env_factory("ctm_test0", lambda config: ctm_test0(config))
R.register_env_factory("ctm_test0_1", lambda config: ctm_test0_1(config))
R.register_env_factory("ctm_test3", lambda config: ctm_test3(config))
R.register_env_factory("ctm_test4", lambda config: ctm_test4(config))
R.register_env_factory("ctm_test5", lambda config: ctm_test5(config))
```

5.5 Miscelaneous scripts

Check is sumo is installed. If not, you might need to specify the sumo path in the configuration file, under "general", or in scripts arguments.

```
python test_sumo.py
```

Check how the ray handles epsilon greedy decaying.

```
python vizu_exp_schedule.py
```

CHAPTER

SIX

CONFIG FILE EXAMPLE

Here is an example `.yaml` file for running an APEX-DQN experiment in grid (synthetic) network environment with the following settings:

- State space: TDTSE (shor-range detection)
- Action space: extend-change, no progression, no masking
- Reward function: queue length

In this example file, we commented on most of the important arguments. For more details about ray's arguments, please refer to labeled ray's relative documentations. For more details about wolf's arguments, please refer to the corresponding code documentation.

Notice: label `# REG <module name>` in some lines represents you can replace that module with any others shown in `wolf.utils.configuration.registry.py`. If you want to replace them, remember to add necessary parameters for the new module. Full describtion of parameters can be found in corresponding files.

This is an example of a configuration file.

```
# arguments related to the experiment, including settings of ray, algorithm,␣
↪environment, etc.
"ray":
  # ray.init(), refer to: https://docs.ray.io/en/master/package-ref.html?
↪highlight=init#ray.init
  "init":
    "local_mode": false
    "log_to_driver": true
    "logging_level": "WARNING"
  "run_experiments":
    "experiments":
      "global_agent":    # name of the experiment

        ####################
        # Trainable algorithm: APEX/DQN/EVALUATOR/...
        # Refer to: https://docs.ray.io/en/master/tune/api_docs/execution.html?
↪highlight=tune.experiments#ray.tune.Experiment
        ####################
        "run": "APEX"
        "checkpoint_freq": 1
        "checkpoint_at_end": true
        "stop":
          "training_iteration": 100

        ####################
        # APEX configuration
        # Refer to: https://docs.ray.io/en/master/rllib-algorithms.html?
↪highlight=algorithms#rllib-algorithms
```

(continues on next page)

(continued from previous page)

```
        # APEX-DQN section
        ####################
        "config":

          ####################
          # OTHERS
          ####################
          "framework": "tf"
          "log_level": "WARNING"

          ####################
          # RL ALGO PARAMS
          ####################
          "num_gpus": 1
          "num_workers": 7    # at least 2 for APEX
          "target_network_update_freq": 100
          "learning_starts": 0
          "timesteps_per_iteration": 1000

          ####################
          # EVALUATION
          ####################
          # "evaluation_interval": 10
          # "evaluation_num_episodes": 10
          # "in_evaluation": False
          # "evaluation_config":
          #   "explore": False
          # "evaluation_num_workers": 0
          # "custom_eval_function": null
          # "use_exec_api": False

          ####################
          # EXPLORATION
          ####################
          # "exploration_config":
          #   "type": "EpsilonGreedy"
          #   "epsilon_schedule":
          #     "type": "ExponentialSchedule"
          #     "schedule_timesteps": 10000
          #     "initial_p": 1.0
          #     "decay_rate": 0.01

          ####################
          # CUSTOM MODEL
          ####################
          "model":
            # identify the custom model here. Replaceable models: refer to
            # wolf.utils.configuration.configuration.py load_custom_models method
            "custom_model": "tdtse"
            "custom_model_config":
              # number of kernels in each CNN layer
              "filters_size": 32
              # size of the final FNN layer
              "dense_layer_size_by_node": 64
              # whether to use progression, this value should be consist with the
              # "use_progression" in "action_params"
              "use_progression": false
```

(continues on next page)

(continued from previous page)

```
    ####################
    # ENVIRONMENT
    ####################
    "gamma": 0.99
    # simulation horizon, if null, horizon will be choosen by env
    "horizon": null
    # environment instance
    "env": "traffic_env_test0"    # REG env_factory
    # environment configuration
    "env_config":
      # the simulator will be used
      "simulator": "traci"
      # simulation related parameters
      "sim_params":
        # whether to restart a simulation upon reset
        "restart_instance": True
        # simulation time step length (unit: s)
        "sim_step": 1
        # whether to print simulation warnings
        "print_warnings": False
        # whether to run with render (SUMO-GUI in this case)
        "render": False
      "env_state_params": null
      "groups_agent_params": null
      # identify how to build the policy mapping
      "multi_agent_config_params":
        # multiple agents share weights or not
        "name": "shared_policy"    # REG multi_agent_config_factory
        "params": {}
      # agent related parameters
      "agents_params":
        # agent type: global or independent
        "name": "global_agent"    # REG agent_factory
        "params":
          # if running an EVALUATOR, identify the policy here
          "default_policy": null    # REG policy
          # whether to use global reward
          "global_reward": false

          ####################
          # CONNECTORS
          # refer to: wolf.environment.traffic.agents.connectors
          ####################

          ### ACTION SPACE
          "action_params":
            "name": "ExtendChangePhaseConnector"    # REG connector
            "params": {}

          ### OBSERVATION SPACE
          "obs_params":
            "name": "TDTSEConnector"    # REG connector
            "params":
              "obs_params":
                "num_history": 60
                "detector_position": [5, 100]
```

(continues on next page)

(continued from previous page)

```
                    "phase_channel": true

                ### REWARD FUNCTION
                "reward_params":
                  "name": "QueueRewardConnector"    # REG connector
                  "params":
                    "stop_speed": 2

# general arguments
"general":
  "id": "main"
  "seed": null
  "repeat": 1
  "is_tensorboardX": false

  # if SUMO_HOME is not in your system path,
  # please identify it here
  # "sumo_home": "/home/ncarrara/sumo_binaries/bin"

  # output file path
  "workspace": "wolf/tests/traffic_env/test0/results"

  "logging":
    "version": 1
    "disable_existing_loggers": false
    "formatters":
      "standard":
        "format": "[%(name)s] %(levelname)s - %(message)s"
    "handlers":
      "default":
        "level": "WARNING"
        "formatter": "standard"
        "class": "logging.StreamHandler"
    "loggers":
      "":
        "handlers": ["default"]
        "level": "WARNING"
        "propagate": false
      "some.logger.you.want.to.enable.in.the.code":
        "handlers": ["default"]
        "level": "ERROR"
        "propagate": false
```

CHAPTER SEVEN

REPRODUCE EXPERIMENTS

7.1 CTM

For all tests:

```
python main.py ../tests/ctm/test$1/global_agent.yaml
```

Where $1 in [0,0_1,2,3,4,5]. For test 2 3 and 4, you can use random_agent.yaml instead.

7.2 TrafficEnv

7.2.1 Synthetic Network (Grid-Net)

For all tests:

```
python main.py ../tests/traffic_env/test$1/$2.yaml
```

Where $1 in [0,0_1,2] and $2 in [global_agent, random, static_min].

Test0 is a single intersection with demand coming from one direction.

Test0_1 is like Test0, but loopdetectors are more upstream, and the demand is platton like.

Test1 has 2 intersections with a uniform inflow, with a master slave configuration, ie on the direction has a lot of demand, while the demand on others directions is very low.

Test2 is like test1 but with 4 intersections.

7.2.2 Real-World Network

Real_net is a environment handling traffic network and demand given by specific files.

You can run a single agent DQN (APEX) on wujiang network (single main intersection) with this command:

```
python main.py ../tests/traffic_env/real_net/dqn.yaml
```

Or run a multi-agent IQL on shenzhen network (3x3 main intersections) with this command:

```
python main.py ../tests/traffic_env/real_net/iql.yaml
```

7.3 QMIX

Until very recently, Ray's QMIX was bugged (it did too much exploration). So we had to work on the original QMIX implementation. So for the moment, testing QMIX follows another pattern:

From the sow45_code directory, follow the below commands:

```
python qmix/src/main.py --config=qmix_traf --env-config=traf
```

Here the configuration has been broken into configuration for the algorithm (`--config`) and configuration for the environment (`--env-config`).

Within `--env-config` we specify `traf` which makes the program read the `traf.yaml` inside the `qmix/src/config/envs/` directory.

From this file one can change the:

- `test_env` to change between different wolf registered environments.
- `render` to render the simulation or not.

And other environment specific configuration properties.

CHAPTER

EIGHT

FUNCTIONAL API

WOLF offer a workflow based on configuration file, run by a single script, main.py. If you need more freedom when designing your WOLF experiment, you can use TrafficEnv as a classic Gym/RLlib environment.

You can instanciate directly TrafficEnv by discarding two parameters: multi_agent_config_params and groups_agent_params. Those arguments are only used in Ray (and Ray's Qmix).

The functional API has not been maintained since a couple now, but you can see an example in wolf/tests/misc/gaussian_flow.py. In this example, we instanciate the subclass "SimpleGridEnv" which is a grid-shape network. To interact with it, the agent feed a dictionary of action (one by agent) using the step function:

```
env.step({"tl_center0": EXTEND})
```

This instruction means there is only one traffic_light, executing action "EXTEND". The return of this step function follows the same structure as the step function of a multi-agents RLLib's environment.