Exploiting Multiple Abstractions in Episodic RL via Reward Shaping

Software Appendix

Installation and execution

Installation

The software is publicly released with a free software licence and it will be installable from GitHub as `pip install git+<url>`. This will also install all the Python dependencies. The only non-Python dependencies are `graphviz`, which we use for visualization and a tool called "Lydia", that we use for converting task specifications into reward functions and Reward Machines (see below).

The instruction we provide here, instead, are for a full development installation, because this procedure guarantees the exact dependencies we have used for running the experiments.

Python version We use Python 3.9. If this version is already installed on the system, the user can skip this paragraph. The 3.9 Python version can be installed with `pyenv`[1]. Please refer to Pyenv installation instructions (note, in particular, the Python build dependencies).

Lydia Lydia is the only non-Python dependency we rely on. Please refer to the package website[2] for installation instructions. Simply, this reduces to executing from a Docker image:

```
docker pull whitemech/lydia:latest
alias lydia='docker run --rm -v$(pwd):/home/default whitemech/lydia lydia "$@"'
```

Note that bash needs single quotes '. They may be wrongly typeset by LaTex in the listing above.

Package dependencies Python dependencies are managed by Poetry[3], that we use to pin and install dependences versions. Please refer to the Python Poetry website for installation instructions. With a working `poetry` installation, it is sufficient to run

[1] `https://github.com/pyenv/pyenv`
[2] `https://github.com/whitemech/lydia`
[3] `https://python-poetry.org/docs/`

```
cd multinav/
poetry env use 3.9
poetry install
```

to install all the Python dependencies at the same version we have used for development. It is advised to do so to avoid dependency issues.

Rllib patch When working with RLlib[4] version 1.13.0, we have encountered a bug in the DQN implementation. We provide a patch file to apply the necessary changes.

```
ENV_PATH=$(poetry env info --path)
RAY_PATH=$ENV_PATH/lib/python3.9/site-packages/ray
cp dqn-fix.patch $(ENV_PATH)/
cd $(ENV_PATH)
patch -i dqn-fix.patch rllib/agents/dqn/dqn.py
```

Make sure to execute these commands after `poetry install` as new installations might revert these changes.

GPU The instructions above work for both CPU and GPU execution. For CPU execution, the user should edit the `run-options.yaml` files to set the `num_gpus` option to 0. If an NVIDIA GPU is present in the system. The user only needs to have the appropriate drivers and CUDA runtime installed.

Execution

Now that the dependency are installed, we can enter the Python environment containing all the dependencies with

```
cd multinav/
poetry shell
```

We assume all the following commands are executed inside this environment.

The program receives command line options. We can receive some help with

```
python -m multinav --help
```

Most of the parameters are provided through configuration files in Yaml format. We provide examples of `run-options.yaml` files in the `inputs` directory of the repository.

To start a training we execute:

```
python -m multinav train --params run-options.yaml
```

To load and execute an already trained agent, we run:

```
python -m multinav test --params <run-options.yaml> --load <checkpoint> [--render]
```

[4] `https://docs.ray.io/en/latest/rllib/index.html`

For example, for a specific run,

```
python -m multinav test --params outputs/office2-1sh/0/logs/run-options.yaml \
--load outputs/office2-1sh/0/models/model_200000.pickle
```

This will load a specific trained agent and perform rollouts for evaluation.

In the source code, the "active" agent refers to $Learner_i^b$ of Algorithm 1. The output is a value function/policy $\hat{\rho}^{b*}$ stored under `models/model_<step>.pickle`. The "passive" agent refers to $Learner_i$, whose estimated optimum $\hat{\rho}^*$ is stored under `models/Extra_<step>.pickle`.

Instructions for reproducing the experiments

The software is a Python 3 package called `multinav`. In input, it receives a configuration file in Yaml format, with all the environment and algorithm hyper-parameters. In output, it generates checkpoints with the agent's parameters and log files with all the evaluation metrics. This section explains the working principles of the software and the content of the attached archive.

We call *run* a single execution of the software from a given seed. With *experiment* we refer to one or multiple runs executed from a single set of hyper-parameters. When training an agent the following directories are created:

```
<experiment-name>/<run-id>/logs/
<experiment-name>/<run-id>/models/
```

The `<experiment-name>` is an identifier for an experiment. For example, the Dueling DQN baseline of Figure 6 of the main paper is called `office2-0dqn`, while our RS approach is under `office2-0sh`. `office2` is the name of an environment and 0 refers to the fact that this environment is the ground MDP $\mathcal{M}_0$. `<run-id>`, instead, is an identifier for the single run. Usually, this is an integer from 0 to 9.

Under `logs` we find these important files:

`experiment.yaml` is the configuration file of an experiment. Inside it, there is the `n-runs` parameter that controls how many runs are associated to this experiment. This file can be used to repeat a new experiment with the same parameters (different seed).

`run-options.yaml` is the configuration file of a single run. This is the file that is actually passed to each execution of the `multinav` software. `run-options.yaml` contains all the algorithm and environment hyper-parameters that are in `experiment.yaml`, plus some details for exact re-execution, such as the seed for this run and the commit hash of the software at the time of execution.

`algorithm-diff.patch` In addition to the commit hash, which is stored in `run-options.yaml`, we also save a patch file. This allow us to keep track, at the time of execution, of any source code changes since the latest commit.

So, to re-execute run number 3 from our reward shaping agent, shown in Figure 2b, we do:

```
cd multinav/
cp -r ../experiments/fig2b/ outputs
git checkout <commit-hash>
git apply outputs/rooms8-1sh/3/logs/algorithm-diff.patch
python -m multinav train --params outputs/rooms8-1sh/3/logs/run-options.yaml
```

This will (over-)write the output files under the same directories. It is possible to modify the keys `logs-dir` and `model-dir` inside `run-options.yaml` to specify different output paths.

Copying the experiment directory under `outputs` is necessary, because some agents might load data from other directories in `outputs`. For example, any reward shaping agent that is learning on some $\mathcal{M}_1$ needs to load the value function from $\mathcal{M}_2$. The git commands guarantee exact execution and needs to be issued at most once per experiment. Finally, the last command starts the training from the same seed and configuration.