S-RL Toolbox Documentation

Antonin Raffin

Apr 05, 2021

Guide

S-RL Toolbox: Reinforcement Learning (RL) and State Representation Learning (SRL) Toolbox for Robotics

Github repository: https://github.com/araffin/robotics-rl-srl

Video: https://youtu.be/qNsHMkIsqJc

This repository was made to evaluate State Representation Learning methods using Reinforcement Learning. It integrates (automatic logging, plotting, saving, loading of trained agent) various RL algorithms (PPO, A2C, ARS, ACKTR, DDPG, DQN, ACER, CMA-ES, SAC, TRPO) along with different SRL methods (see SRL Repo) in an efficient way (1 Million steps in 1 Hour with 8-core cpu and 1 Titan X GPU).

We also release customizable Gym environments for working with simulation (Kuka arm, Mobile Robot in PyBullet, running at 250 FPS on a 8-core machine) and real robots (Baxter Robot, Robobo with ROS).

CHAPTER 1

Main Features

- 10 RL algorithms (Stable Baselines included)
- logging / plotting / visdom integration / replay trained agent
- hyperparameter search (hyperband, hyperopt)
- integration with State Representation Learning (SRL) methods (for feature extraction)
- visualisation tools (explore latent space, display action proba, live plot in the state space, . . .)
- robotics environments to compare SRL methods
- easy install using anaconda env or Docker images (CPU/GPU)

Related papers:

- "Decoupling feature extraction from policy learning: assessing benefits of state representation learning in goal based robotics" (Raffin et al. 2018) https://arxiv.org/abs/1901.08651
- "S-RL Toolbox: Environments, Datasets and Evaluation Metrics for State Representation Learning" (Raffin et al., 2018) https://arxiv.org/abs/1809.09369

Note: This documentation only gives an overview of the RL Toolbox, and provides some examples. However, for a complete list of possible argument, you have to use the `--help` argument. For example, you can try: `python -m rl_baselines.train --help`

1.1 Installation

Python 3 is required (python 2 is not supported because of OpenAI baselines)

Note: we are using Stable Baselines, a fork of OpenAI Baselines with unified interface and other improvements (e.g. tensorboard support).

1.1.1 Using Anaconda

0. Download the project (note the `--recursive` argument because we are using git submodules):

```
git clone git@github.com:araffin/robotics-rl-srl.git --recursive
```

1. Install the swig library:

```
sudo apt-get install swig
```

2. Install the dependencies using `environment.yml` file (for anaconda users) in the current environment

```
conda env create --file environment.yml
source activate py35
```

PyBullet Documentation

1.1.2 Using Docker

Use Built Images

GPU image (requires nvidia-docker):

```
docker pull araffin/rl-toolbox
```

CPU only:

```
docker pull araffin/rl-toolbox-cpu
```

Build the Docker Images

Build GPU image (with nvidia-docker):

```
docker build . -f docker/Dockerfile.gpu -t rl-toolbox
```

Build CPU image:

```
docker build . -f docker/Dockerfile.cpu -t rl-toolbox-cpu
```

Note: if you are using a proxy, you need to pass extra params during build and do some tweaks:

```
--network=host --build-arg HTTP_PROXY=http://your.proxy.fr:8080/ --build-arg http_
↪proxy=http://your.proxy.fr:8080/ --build-arg HTTPS_PROXY=https://your.proxy.fr:8080/
↪ --build-arg https_proxy=https://your.proxy.fr:8080/
```

Run the images

Run the nvidia-docker GPU image

```
docker run -it --runtime=nvidia --rm --network host --ipc=host --name test --mount␣
↪src="$(pwd)",target=/tmp/rl_toolbox,type=bind araffin/rl-toolbox bash -c 'source␣
↪activate py35 && cd /tmp/rl_toolbox/ && python -m rl_baselines.train --srl-model␣
↪ground_truth --env MobileRobotGymEnv-v0 --no-vis --num-timesteps 1000'
```

Or, with the shell file:

```
./run_docker_gpu.sh python -m rl_baselines.train --srl-model ground_truth --env␣
↪MobileRobotGymEnv-v0 --no-vis --num-timesteps 1000
```

Run the docker CPU image

```
docker run -it --rm --network host --ipc=host --name test --mount src="$(pwd)",
↪target=/tmp/rl_toolbox,type=bind araffin/rl-toolbox-cpu bash -c 'source activate␣
↪py35 && cd /tmp/rl_toolbox/ && python -m rl_baselines.train --srl-model ground_
↪truth --env MobileRobotGymEnv-v0 --no-vis --num-timesteps 1000'
```

Or, with the shell file:

```
./run_docker_cpu.sh python -m rl_baselines.train --srl-model ground_truth --env␣
↪MobileRobotGymEnv-v0 --no-vis --num-timesteps 1000
```

Explanation of the docker command:

- `docker run -it` create an instance of an image (=container), and run it interactively (so ctrl+c will work)
- `--rm` option means to remove the container once it exits/stops (otherwise, you will have to use `docker rm`)
- `--network host` don't use network isolation, this allow to use visdom on host machine
- `--ipc=host` Use the host system's IPC namespace. It is needed to train SRL model with PyTorch. IPC (POSIX/SysV IPC) namespace provides separation of named shared memory segments, semaphores and message queues.
- `--name test` give explicitely the name `test` to the container, otherwise it will be assigned a random name
- `--mount src=...` give access of the local directory (`pwd` command) to the container (it will be map to `/tmp/rl_toolbox`), so all the logs created in the container in this folder will be kept (for that you need to pass the `--log-dir logs/` option)
- `bash -c 'source activate py35 && ...` Activate the conda enviroment inside the docker container, and launch an experiment (`python -m rl_baselines.train ...`)

1.2 Getting Started

Here is a quick example of how to train a PPO2 agent on `MobileRobotGymEnv-v0` environment for 10 000 steps using 4 parallel processes:

```
python -m rl_baselines.train --algo ppo2 --no-vis --num-cpu 4 --num-timesteps 10000 --
↪env MobileRobotGymEnv-v0
```

The complete command (logs will be saved in *logs/* folder):

```
python -m rl_baselines.train --algo rl_algo --env env1 --log-dir logs/ --srl-model␣
↪raw_pixels --num-timesteps 10000 --no-vis
```

To use the robot's position as input instead of pixels, just pass `--srl-model ground_truth` instead of `--srl-model raw_pixels`

1.3 Reinforcement Learning

Note: All CNN policies normalize input, dividing it by 255. By default, observations are not stacked. For SRL, states are normalized using a running mean/std average.

About frame-stacking, action repeat (frameskipping) please read this blog post: Frame Skipping and Pre-Processing for DQN on Atari

Before you start a RL experiment, you have to make sure that a visdom server is running, unless you deactivate visualization.

Launch visdom server:

```
python -m visdom.server
```

1.3.1 RL Algorithms: OpenAI Baselines and More

Several algorithms from Stable Baselines have been integrated along with some evolution strategies and SAC:

- A2C: A synchronous, deterministic variant of Asynchronous Advantage Actor Critic (A3C).
- ACER: Sample Efficient Actor-Critic with Experience Replay
- ACKTR: Actor Critic using Kronecker-Factored Trust Region
- ARS: Augmented Random Search (https://arxiv.org/abs/1803.07055)
- CMA-ES: Covariance Matrix Adaptation Evolution Strategy
- DDPG: Deep Deterministic Policy Gradients
- DeepQ: DQN and variants (Double, Dueling, prioritized experience replay)
- PPO1: Proximal Policy Optimization (MPI Implementation)
- PPO2: Proximal Policy Optimization (GPU Implementation)
- SAC: Soft Actor Critic
- TRPO: Trust Region Policy Optimization (MPI Implementation)

Train an Agent with Discrete Actions

To train an agent (without visualization with visdom):

```
python -m rl_baselines.train --algo ppo2 --log-dir logs/ --no-vis
```

You can train an agent on the latest learned model (knowing it's type) located at `log_folder: srl_zoo/logs/DatasetName/` (defined for each environment in `config/srl_models.yaml`):

```
python -m rl_baselines.train --algo ppo2 --log-dir logs/ --latest --srl-model srl_
↪combination --env MobileRobotGymEnv-v0
```

Train an Agent with Continuous Actions

Continuous actions have been implemented for DDPG, PPO2, ARS, CMA-ES, SAC and random agent. To use continuous actions in the position space:

```
python -m rl_baselines.train --algo ppo2 --log-dir logs/ -c
```

To use continuous actions in the joint space:

```
python -m rl_baselines.train --algo ppo2 --log-dir logs/ -c -joints
```

Train an agent multiple times on multiple environments, using different methods

To run multiple enviroments with multiple SRL models for a given algorithm (you can use the same arguments as for training should you need to specify anything to the training script):

```
python  -m rl_baselines.pipeline --algo ppo2 --log-dir logs/ --env env1 env2 [...] --
↪srl-model model1 model2 [...]
```

For example, run a total of 30 experiments of ppo2 with 4 cpus and randomly initialized target position, in the default environment using VAE, and using ground truth (15 experiments each):

```
python  -m rl_baselines.pipeline --algo ppo2 --log-dir logs/ --srl-model vae ground_
↪truth --random-target --num-cpu 4 --num-iteration 15
```

Load a Trained Agent

To load a trained agent and see the result:

```
python -m replay.enjoy_baselines --log-dir path/to/trained/agent/ --render
```

1.3.2 Add your own RL algorithm

1. Create a class that inherits `rl_baselines.base_classes.BaseRLObject` which implements your algorithm. You will need to define specifically:
   - `save(save_path, _locals=None)`: to save your model during or after training.
   - `load(load_path, args=None)`: to load and return a saved instance of your class (static function).
   - `customArguments(parser)`: `@classmethod` to define specifics command line arguments from `train.py` or `pipeline.py` calls, then returns the parser object.
   - `getAction(observation, dones=None)`: to get the action from a given observation.
   - `makeEnv(self, args, env_kwargs=None, load_path_normalise=None)`: override if you need to change the environment wrappers (static function).
   - `train(args, callback, env_kwargs=None, hyperparam=None)`: to create the environment, and train your algorithm on said environment.
   - (OPTIONAL) `getActionProba(observation, dones=None)`: to get the action probabilities from a given observation. This is used for the action probability plotting in `replay.enjoy_baselines`.
   - (OPTIONAL) `getOptParam()`: `@classmethod` to return the hyperparameters that can be optimised through the callable argument. Along with the type and range of said parameters.
2. Add your class to the `registered_rl` dictionary in `rl_baselines/registry.py`, using this format `NAME: (CLASS, ALGO_TYPE, [ACTION_TYPE])`, where:

- `NAME`: is your algorithm's name.
- `CLASS`: is your class that inherits `BaseRLObject`.
- `ALGO_TYPE`: is the type of algorithm, defined by the enumerator `AlgoType` in `rl_baselines/__init__.py`, can be `REINFORCEMENT_LEARNING`, `EVOLUTION_STRATEGIES` or `OTHER` (`OTHER` is used to define algorithms that can't be run in `enjoy_baselines.py` (ex: Random_agent)).
- `[ACTION_TYPE]`: is the list of compatible type of actions, defined by the enumerator `ActionType` in `rl_baselines/__init__.py`, can be `CONTINUOUS` and/or `DISCRETE`.

3. Now you can call your algorithm using `--algo NAME` with `train.py` or `pipeline.py`.

1.4 Hyperparameter Search

This repository also allows hyperparameter search, using hyperband or hyperopt for the implemented RL algorithms

for example, here is the command for a hyperband search on PPO2, ground truth on the mobile robot environment:

```
python -m rl_baselines.hyperparam_search --optimizer hyperband --algo ppo2 --env␣
↪MobileRobotGymEnv-v0 --srl-model ground_truth
```

1.5 Environments

All the environments we propose follow the OpenAI Gym interface. We also extended this interface (adding extra methods) to work with SRL methods (see *State Representation Learning Models*).

OpenAI Gym repo: https://github.com/openai/gym

1.5.1 Available environments

You can find a recap table in the README.

- Kuka arm: Here we have a Kuka arm which must reach a target, here a button.
  - KukaButtonGymEnv-v0: Kuka arm with a single button in front.
  - KukaRandButtonGymEnv-v0: Kuka arm with a single button in front, and some randomly positioned objects
  - Kuka2ButtonGymEnv-v0: Kuka arm with 2 buttons next to each others, they must be pressed in the correct order (lighter button, then darker button).
  - KukaMovingButtonGymEnv-v0: Kuka arm with a single button in front, slowly moving left to right.
- Mobile robot: Here we have a mobile robot which reach a target position
  - MobileRobotGymEnv-v0: A mobile robot on a 2d terrain where it needs to reach a target position (yellow cylinder).
  - MobileRobot2TargetGymEnv-v0: A mobile robot on a 2d terrain where it needs to reach two target positions, in the correct order (yellow target, then red target).
  - MobileRobot1DGymEnv-v0: A mobile robot on a 1d slider where it can only go up and down, it must reach a target position.
  - MobileRobotLineTargetGymEnv-v0: A mobile robot on a 2d terrain where it needs to reach a colored band going across the terrain.

- Racing car: Here we have the interface for the Gym racing car environment. It must complete a racing course in the least time possible (only available in a terminal with X running)
  - CarRacingGymEnv-v0: A racing car on a racing course, it must complete the racing course in the least time possible.
- Baxter: A baxter robot that must reach a target, with its arms.
  - Baxter-v0: A bridge to use a baxter robot with ROS (in simulation, it uses Gazebo)
- Robobo: A Robobo robot that must reach a target position.
  - RoboboGymEnv-v0: A bridge to use a Robobo robot with ROS.
- Omnidirectional Robot: A mobile robot on a 2d terrain that must reach a target position.
  - OmnirobotEnv-v0: A bridge to use a real Omnirobot with ROS (also available in simulation).

1.5.2 Generating Data

To test the environment with random actions:

```
python -m environments.dataset_generator --no-record-data --display
```

Can be as well used to render views (or dataset) with two cameras if `multi_view=True`.

To **record data** (i.e. generate a dataset) from the environment for **training a SRL model**, using random actions:

```
python -m environments.dataset_generator --num-cpu 4 --name folder_name
```

1.5.3 Add a custom environment

1. Create a class that inherits `environments.srl_env.SRLGymEnv` which implements your environment. You will need to define specifically:
   - `getTargetPos()`: returns the position of the target.
   - `getGroundTruthDim()`: returns the number of dimensions used to encode the ground truth.
   - `getGroundTruth()`: returns the ground truth state.
   - `step(action)`: step the environment in simulation with the given action.
   - `reset()`: re-initialise the environment.
   - `render(mode='human')`: returns an observation of the environment.
   - `close()`: closes the environment, override if you need to change it.
   - Make sure `__init__` has the parameter `**_kwargs` in order to ignore useless flag parameters sent by the calling code.
2. Add this code to the same file as the class declaration

```
def getGlobals():
    """
    :return: (dict)
    """
    return globals()
```

it will allow the logging of constant values used by the class

3. Add your class to the `registered_env` dictionary in `environments/registry.py`, using this format `NAME: (CLASS, SUPER_CLASS, PLOT_TYPE, THREAD_TYPE)`, where:
   - `NAME`: is your environment's name, it must only contain `[A-Z][a-z][0-9]` and end with the version number in this format: `-v{number}`.
   - `CLASS`: is your class that is a subclass of `SRLGymEnv`.
   - `SUPER_CLASS`: is the super class of your class, this is for saving all the globals and parameters.
   - `PLOT_TYPE`: is the type of plotting for `replay.enjoy_baselines`, defined by the enumerator `PlottingType` in `environments/__init__.py`, can be `PLOT_2D` or `PLOT_3D` (use `PLOT_3D` if unsure).
   - `THREAD_TYPE`: is the type of multithreading supported by the environment, defined by the enumerator `ThreadingType` in `environments/__init__.py`, can be (from most restricive to less restricive) `PROCESS`, `THREAD` or `NONE` (use `NONE` if unsure).
4. Add the name of the environment to `config/srl_models.yaml`, with the location of the saved model for each SRL model (can point to a dummy location, but must be defined).
5. Now you can call your environment using `--env NAME` with `train.py`, `pipeline.py` or `dataset_generator.py`.

1.6 State Representation Learning Models

A State Representation Learning (SRL) model aims to compress from a high dimensional observation, a compact representation. This learned representation can be used instead of learning a policy directly from pixels, in a deep reinforcement learning algorithm.

A more detailed overview: https://arxiv.org/pdf/1802.04181.pdf

Please look the SRL Repo to learn how to train a state representation model. Then you must edit `config/srl_models.yaml` and set the right path to use the learned state representations.

To train a Reinforcement learning agent on a specific SRL model:

```
python -m rl_baselines.train --algo ppo2 --log-dir logs/ --srl-model model_name
```

1.6.1 Available SRL models

The available state representation models are:

- ground_truth: Hand engineered features (e.g., robot position + target position for mobile robot env)
- raw_pixels: Learning a policy in an end-to-end manner, directly from pixels to actions.
- supervised: A model trained with Ground Truth states as targets in a supervised setting.
- autoencoder: an autoencoder from the raw pixels
- vae: a variational autoencoder from the raw pixels
- inverse: an inverse dynamics model
- forward: a forward dynamics model
- srl_combination: a model combining several losses (e.g. vae + forward + inverse...) for SRL
- pca: pca applied to the raw pixels

- robotic_priors: robotic priors model (Learning State Representations with Robotic Priors)
- multi_view_srl: a SRL model using views from multiple cameras as input, with any of the above losses (e.g triplet and others)
- joints: the arm's joints angles (kuka environments only)
- joints_position: the arm's x,y,z position and joints angles (kuka environments only)

Note: For debugging, we integrated logging of states (we save the states that the RL agent encountered during training) with SAC algorithm. To log the states during RL training you have to pass the `--log-states` argument:

```
python -m rl_baselines.train --srl-model ground_truth --env␣
↪MobileRobotLineTargetGymEnv-v0 --log-dir logs/ --algo sac --reward-scale 10 --log-
↪states
```

The states will be saved in a `log_srl/` folder as numpy archives, inside the log folder of the rl experiment.

1.6.2 Add a custom SRL model

If your SRL model is a charateristic of the environment (position, angles, ...):

1. Add the name of the model to the `registered_srl` dictionary in `state_representation/registry.py`, using this format `NAME: (SRLType.ENVIRONMENT, [LIMITED_TO_ENV])`, where:
   - `NAME`: is your model's name.
   - `[LIMITED_TO_ENV]`: is the list of environments where this model works (will check for subclass), set to `None` if this model applies to every environment.
2. Modifiy the `def getSRLState(self, observation)` in the environments to return the data you want for this model.
3. Now you can call your SRL model using `--srl-model NAME` with `train.py` or `pipeline.py`.

Otherwise, for the SRL model that are external to the environment (Supervised, autoencoder, ...):

1. Add your SRL model that inherits `SRLBaseClass`, to the function `state_representation.models.loadSRLModel`.
2. Add the name of the model to the `registered_srl` dictionary in `state_representation/registry.py`, using this format `NAME: (SRLType.SRL, [LIMITED_TO_ENV])`, where:
   - `NAME`: is your model's name.
   - `[LIMITED_TO_ENV]`: is the list of environments where this model works (will check for subclass), set to `None` if this model applies to every environment.
3. Add the name of the model to `config/srl_models.yaml`, with the location of the saved model for each environment (can point to a dummy location, but must be defined).
4. Now you can call your SRL model using `--srl-model NAME` with `train.py` or `pipeline.py`.

1.7 Plotting

1.7.1 Plot Learning Curve

To plot a learning curve from logs in visdom, you have to pass path to the experiment log folder:

```
python -m replay.plots --log-dir /logs/raw_pixels/ppo2/18-03-14_11h04_16/
```

To aggregate data from different experiments (different seeds) and plot them (mean + standard error). You have to pass path to rl algorithm log folder (parent of the experiments log folders):

```
python -m replay.aggregate_plots --log-dir /logs/raw_pixels/ppo2/ --shape-reward --
↪timesteps --min-x 1000 -o logs/path/to/output_file
```

Here it plots experiments with reward shaping and that have a minimum of 1000 data points (using timesteps on the x-axis), the plot data will be saved in the file `output_file.npz`.

To create a comparison plots from saved plots (.npz files), you need to pass a path to folder containing .npz files:

```
python -m replay.compare_plots -i logs/path/to/folder/ --shape-reward --timesteps
```

1.7.2 Gather Results

Gather results for all experiments of an enviroment. It will report mean performance for a given budget.

```
python -m replay.gather_results -i path/to/envdir/ --min-timestep 5000000 --timestep-
↪budget 1000000 2000000 3000000 5000000 --episode-window 100
```

1.8 Working With Real Robots: Baxter, Robobo and Omnirobot

1.8.1 Baxter Robot with Gazebo and ROS

Gym Wrapper for baxter environment, more details in the dedicated README (environments/gym_baxter/README.md).

Warning: ROS (and Gazebo + Baxter) only works with python2, whereas this repo (except the ROS scripts) works with python3. For Ros/Baxter installation, please look at the Official Tutorial. Also, ROS comes with its own version of OpenCV, so when running the python3 scripts, you need to deactivate ROS. In the same vein, if you use Anaconda, you need to disable it when you want to run ROS scripts (denoted as python 2 in the following instructions).

0. Download ROS packages (ROS kinetic) and install them in your catkin workspace:
   - arm scenario experiments, branch "rl"
   - arm scenario simulator branch kinetic-devel
1. Start ros nodes (Python 2):

```
roslaunch arm_scenario_simulator baxter_world.launch
rosrun arm_scenario_simulator spawn_objects_example

python -m real_robots.gazebo_server
```

Then, you can either try to teleoperate the robot (python 3):

```
python -m real_robots.teleop_client
```

or test the environment with random actions (using the gym wrapper):

```
python -m environments.gym_baxter.test_baxter_env
```

If the port is already used, you can see the program pid using the following command:

```
sudo netstat -lpn | grep :7777
```

and then kill it (with `kill -9 program_pid`)

or in one line:

```
kill -9 `sudo lsof -t -i:7777`
```

1.8.2 Working With a Real Baxter Robot

WARNING: Please read COMPLETELY the following instructions before running and experiment on a real baxter.

Recording Data With a Random Agent for SRL

1. Change you environment to match baxter ROS settings (usually using the `baxter.sh` script from RethinkRobotics) or in your .bashrc:

```
# NB: This is only an example
export ROS_HOSTNAME=192.168.0.211  # Your IP
export ROS_MASTER_URI=http://baxter.local:11311 # Baxter IP
```

2. Calibrate the different values in `real_robots/constants.py` using `real_robots/real_baxter_debug.py`:

- Set USING_REAL_BAXTER to True
- Position of the target: BUTTON_POS
- Init position and orientation: LEFT_ARM_INIT_POS, LEFT_ARM_ORIENTATION
- Position of the table (minimum z): Z_TABLE
- Distance below which the target is considered to be reached: DIST_TO_TARGET_THRESHOLD
- Distance above which the agent will get a negative reward: MAX_DISTANCE
- Maximum number of steps per episode: MAX_STEPS

3. Configure images topics in `real_robots/constants.py`:

- IMAGE_TOPIC: main camera
- SECOND_CAM_TOPIC: second camera (set it to None if you don't want to use a second camera)
- DATA_FOLDER_SECOND_CAM: folder where the images of the second camera will be saved

4. Launch ROS bridge server (python 2):

```
python -m real_robots.real_baxter_server
```

5. Deactivate ROS from your environment and switch to python 3 environment (for using this repo)
6. Set the number of episodes you want to record, name of the experiment and random seed in `environments/gym_baxter/test_baxter_env.py`
7. Record data using a random agent:

```
python -m environments.gym_baxter.test_baxter_env
```

8. Wait until the end... Note: the real robot runs at approximately 0.6 FPS.

NB: If you want to save the image without resizing, you need to comment the line in the method `getObservation()` in `environments/gym_baxter/baxter_env.py`

RL on a Real Baxter Robot

1. Update the settings in `rl_baselines/train.py`, so it saves and log the training more often (LOG_INTERVAL, SAVE_INTERVAL, ...)
2. Make sure that USING_REAL_BAXTER is set to True in `real_robots/constants.py`.
3. Launch ROS bridge server (python 2):

```
python -m real_robots.real_baxter_server
```

4. Start visdom for visualizing the training

```
python -m visdom.server
```

4. Train the agent (python 3)

```
python -m rl_baselines.train --srl-model ground_truth --log-dir logs_real/ --num-
↪stack 1 --shape-reward --algo ppo2 --env Baxter-v0
```

1.8.3 Working With a Real Robobo

Robobo Documentation

Note: the Robobo is controlled using time (the feedback frequency is too low to do closed-loop control) The robot was calibrated for a constant speed of 10.

Recording Data With a Random Agent for SRL

1. Change you environment to match Robobo ROS settings or in your .bashrc: NOTE: Robobo is using ROS Java, if you encounter any problem with the cameras (e.g. with a xtion), you should create the master node on your computer and change the settings in the robobo dev app.

```
# NB: This is only an example
export ROS_HOSTNAME=192.168.0.211  # Your IP
export ROS_MASTER_URI=http://robobo.local:11311 # Robobo IP
```

2. Calibrate the different values in `real_robots/constants.py` using `real_robots/real_robobo_server.py` and `real_robots/teleop_client.py` (Client for teleoperation):

- Set USING_ROBOBO to True
- Area of the target: TARGET_INITIAL_AREA
- Boundaries of the enviroment: (MIN_X, MAX_X, MIN_Y, MAX_Y)

- Maximum number of steps per episode: MAX_STEPS IMPORTANT NOTE: if you use color detection to detect the target, you need to calibrate the HSV thresholds `LOWER_RED` and `UPPER_RED` in `real_robots/constants.py` (for instance, using this script). Be careful, you may have to change the color conversion (`cv2.COLOR_BGR2HSV` instead of `cv2.COLOR_RGB2HSV`)

3. Configure images topics in `real_robots/constants.py`:

- IMAGE_TOPIC: main camera
- SECOND_CAM_TOPIC: second camera (set it to None if you don't want to use a second camera)
- DATA_FOLDER_SECOND_CAM: folder where the images of the second camera will be saved

NOTE: If you want to use robobo's camera (phone camera), you need to republish the image to the raw format:

```
rosrun image_transport republish compressed in:=/camera/image raw out:=/camera/image_
↪repub
```

4. Launch ROS bridge server (python 2):

```
python -m real_robots.real_robobo_server
```

5. Deactivate ROS from your environment and switch to python 3 environment (for using this repo)
6. Set the number of episodes you want to record, name of the experiment and random seed in `environments/robobo_gym/test_robobo_env.py`
7. Record data using a random agent:

```
python -m environments.robobo_gym.test_robobo_env
```

8. Wait until the end... Note: the real robobo runs at approximately 0.1 FPS.

NB: If you want to save the image without resizing, you need to comment the line in the method `getObservation()` in `environments/robobo_gym/robobo_env.py`

RL on a Real Robobo

1. Update the settings in `rl_baselines/train.py`, so it saves and logs the training more often (LOG_INTERVAL, SAVE_INTERVAL, ...)
2. Make sure that USING_ROBOBO is set to True in `real_robots/constants.py`.
3. Launch ROS bridge server (python 2):

```
python -m real_robots.real_robobo_server
```

4. Start visdom for visualizing the training

```
python -m visdom.server
```

4. Train the agent (python 3)

```
python -m rl_baselines.train --srl-model ground_truth --log-dir logs_real/ --num-
↪stack 1 --algo ppo2 --env RoboboGymEnv-v0
```

1.8.4 Working With a Real Omnirobot

By default, Omnirobot uses the same reward and terminal policy with the MobileRobot environment. Thus each episodes will have exactly 251 steps, and when the robot touches the target, it will get `reward=1`, when it touches the border, it will get `reward=-1`, otherwise, `reward=0`.

All the important parameters are writed in constants.py, thus you can simply modified the reward or terminal policy of this environment.

Architecture of Omnirobot

Architecture of Real Omnirobot

The omnirobot's environment contains two principle components (two threads).

- `real_robots/omnirobot_server.py` (python2, using ROS to communicate with robot)
- `environments/omnirobot_gym/omnirobot_env.py` (python3, wrapped baseline environment)

These two components uses zmq socket to communicate. The socket port can be changed, and by defualt it's 7777. These two components should be launched manually, because they use different environment (ROS and anaconda).

Architecture of Omnirobot Simulator

The simulator has only one thread, omnirobot_env. The simulator is a object of this running thread, it uses exactly the same api as `zmq`, thus `omnirobot_server` can be easily switched to `omnirobot_simulator_server` without changing code of `omnirobot_env`.

Switch between real robot and simulator

- Switch from real robot to simulator modify `real_robots/constants.py`, set `USING_OMNIROBOT = False` and `USING_OMNIROBOT_SIMULATOR = True`
- Switch from simulator to real robot: modify `real_robots/constants.py`, set `USING_OMNIROBOT = True` and `USING_OMNIROBOT_SIMULATOR = False`

Real Omnirobot

Omnirobot offers the clean environment for RL, for each step of RL, the real robot does a close-loop positional control to reach the supposed position.

When the robot is moving, `omnirobot_server` will be blocked until it receives a msg from the topic `finished`, which is sent by the robot. This blocking has a time out (by default 30s), thus if anything unexpected happens, the `omnirobot_server` will fail and close.

Launch RL on real omnirobot

To launch the rl experience of omnirobot, do these step-by-step:

- switch to real robot (modify constans.py, ensure `USING_OMNIROBOT = True`)
- setup ROS environment and comment `anaconda` in `~/.bashrc`, launch a new terminal, run

```
python -m real_robots.omnirobot_server
```

- comment ROS environment and uncomment `anaconda` in `~/.bashrc`, launch a new terminal.
- If you want to train RL on real robot, run (with other options customizable):

```
python -m rl_baselines.train --env OmnirobotEnv-v0
```

- If you want to replay the RL policy on real robot, which can be trained on the simulator, run:

```
python -m replay.enjoy_baselines --log-dir path/to/RL/logs -render
```

Recording Data of real omnirobot

To launch a acquisition of real robot's dataset, do these step-by-step:

- switch to real robot (modify constans.py, ensure `USING_OMNIROBOT = True`)
- setup ROS environment and comment *anaconda* in *~/.bashrc*, launch a new terminal, run:

```
python -m real_robots.omnirobot_server
```

- Change `episodes` to the number of you want in `environments/omnirobot_gym/test_env.py`
- comment ROS environment and uncomment `anaconda` in `~/.bashrc`, launch a new terminal, run:

```
python -m environments.omnirobot_gym.test_env
```

Note that you should move the target manually between the different episodes. Attention, you can try to use Random Agent or a agent always do the toward target policy (this can increase the positive reward proportion in the dataset), or combine them by setting a proportion (`TORWARD_TARGET_PROPORTION`).

Omnirobot Simulator

This simulator uses photoshop tricks to make realistic image of environment. It need several image as input:

- back ground image (480x480, undistorted)
- robot's tag/code, cropped from a real environment image(480x480, undistorted), with a margin 3 or 4 pixels.
- target's tag/code, cropped from a real environment image (480x480, undistorted), with a margin 3 or 4 pixels.

It also needs some important information:

- margin of markerts
- camera info file's path, which generated by ROS' camera_calibration package.

The camera matrix should be corresponding with original image size (eg. 640x480 for our case)

The detail of the inputs above can be find from OmniRobotEnvRender's comments.

Noise of Omnirobot Simulator

To make the simulator more general, and make RL/SRL more stable, several types of noise are added to it. The parameters of these noises can be modified from the top of `omnirobot_simulator_server.py`

- noise of robot position, yaw. Gaussian noise, controlled by `NOISE_VAR_ROBOT_POS` and `NOISE_VAR_ROBOT_YAW`.
- noise of markers in pixel-wise. Gaussian noise to simulate camera's noise, apply pixel-wise noise on the markers' images, controlled by `NOISE_VAR_TARGET_PIXEL` and `NOISE_VAR_ROBOT_PIXEL`.

- noise of environment's luminosity. Apply Gaussian noise on LAB space of output image, to simulate the environment's luminosity change, controlled by `NOISE_VAR_ENVIRONMENT`.
- noise of marker's size. Change size of robot's and target's marker proportionally, to simulate the position variance on the vertical axis. This kind of noise is controlled by `NOISE_VAR_ROBOT_SIZE_PROPOTION` and `NOISE_VAR_TARGET_SIZE_PROPOTION`.

Known issues of Omnirobot

- **The script `omnirobot_server.py` in robotics-rl-srl cannot be simply quitted by ctrl-c.**
  - This is because the zmq in python2 uses blocking behavior, even SIGINT cannot be detected when it is blocking.
  - To quit the program, you should send `SIGKILL` to it. This can be done by `kill -9` or `htop`.
- **Error: `ImportError: /opt/ros/kinetic/lib/python2.7/dist-packages/cv2.so: undefined sym`**
  - You probably run a program expected to run in `conda` environment, sometimes even `~/.bashrc` is changed, and correctly applies `source ~/.bashrc`, the environment still stays with `ros`.
  - In this situation, simply re-check the contents in `~/.bashrc`, and open another new terminal to launch the programme.
- **Stuck at `wait for client to connect` or `waiting to connect server`, there are several possible reasons.**
  - Port for client and server are not same. Try to use the same one
  - Port is occupied by another client/server, you should kill it. If you cannot find the process which occupies this port, use `fuser 7777\tcp -k` to kill it directly. (7777 can be changed to any number of port).

1.9 Running Tests

Download the test datasets kuka_gym_test and kuka_gym_dual_test and put it in `srl_zoo/data/` folder.

```
./run_tests.sh --all
```

1.10 Changelog

For download links, please look at Github release page.

1.10.1 Release 1.3.0 (2019-??-??)

- added OmniRobot environment for simulation and real life setting
- updated package version (gym, stable-baselines)
- updated doc and tests
- added script for merging datasets

1.10.2 Release 1.2.0 (2019-01-17)

- fixed a bug in the dataset generator where the GUI was instantiated two times
- updated stable-baselines version + srl-zoo submodule
- add stable-baselines SAC version
- remove pytorch SAC version **breaking changes**

1.10.3 Release 1.0 (2018-10-09)

Stable Baselines Version

Model trained with previous release are not compatible with this version.

- refactored all rl baselines to integrate with Stable Baselines **breaking changes**
- updated plotting scripts
- added doc

1.10.4 Release 0.4 (2018-09-25)

First Stable Version

Initial release, using OpenAI Baselines (and patches) for the RL algorithms.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search