Digideep Documentation

Release 2019

Mohammadreza Sharif

Jun 23, 2021

Notes:

Digideep is a pipeline for fast prototyping Deep Reinforcement Learning (DeepRL) algorithms which uses PyTorch and Gym / dm_control.

Some important features of Digideep are:

digideep.main.**entrypoint**()

digideep.main.**main**(*session*)

CHAPTER 1

Installation

1.1 Requirements

- Python 3
- PyTorch
- [OPTIONAL] Tensorboard.
- MuJoCo `v200`.
- mujoco_py and Gym.
- dm_control.

Note: If you are a student, you can get a free student license for MuJoCo.

1.2 Installation

Simply download the package using the following command and add it to your `PYTHONPATH`:

1.3 Set your environment

Add the following to your `.bashrc` or `.zshrc`:

```
# Assuming you have installed mujoco in '$HOME/.mujoco'
export LD_LIBRARY_PATH=$HOME/.mujoco/mujoco200_linux/bin:$LD_LIBRARY_PATH
export MUJOCO_GL=glfw
```

1.4 Patch `dm_control` initialization issue

If you hit an error regarding GLFW initialization, try the following patch:

Go to the `digideep` installation path and run:

```
cd <digideep_path>
cp patch/glfw_renderer.py `pip show dm_control | grep -Po 'Location: (\K.*)'`/dm_
↪control/_render
```

CHAPTER 2

Usage

2.1 Training/Replaying

Listing 1: Command-line arguments

```
$ python -m digideep.main --help
usage: main.py [-h] [--load-checkpoint <path>] [--play]
               [--session-path <path>] [--save-modules <path> [<path> ...]]
               [--log-level <n>] [--visdom] [--visdom-port <n>]
               [--monitor-cpu] [--monitor-gpu] [--params <name>]
               [--cpanel <json dictionary>]

optional arguments:
    -h, --help            show this help message and exit
    --load-checkpoint <path>
                          Load a checkpoint to resume training from that point.
    --play                Will play the stored policy.
    --session-path <path>
                          The path to store the sessions. Default is in /tmp
    --save-modules <path> [<path> ...]
                          The modules to be stored in the session.
    --log-level <n>       The logging level: 0 (debug and above), 1 (info and
                          above), 2 (warn and above), 3 (error and above), 4
                          (fatal and above)
    --visdom              Whether to use visdom or not!
    --visdom-port <n>     The port of visdom server, it's on 8097 by default.
    --monitor-cpu         Use to monitor CPU resource statistics on Visdom.
    --monitor-gpu         Use to monitor GPU resource statistics on Visdom.
    --params <name>       Choose the parameter set.
    --cpanel <json dictionary>
                          Set the parameters of the cpanel by a json dictionary.
```

Listing 2: Example Usage

```
# Start a training session for a MuJoCo environment using DDPG
# Default environment is "Pendulum-v0"
python -m digideep.main --params digideep.params.classic_ddpg

# Start a training session for an Atari environment using PPO
# Default environment is "PongNoFrameskip-v4"
python -m digideep.main --params digideep.params.atari_ppo

# Start a training session for a MuJoCo environment using PPO
# Default environment is "Ant-v2"
python -m digideep.main --params digideep.params.mujoco_ppo

# Change the parameters in command-line
python -m digideep.main --params digideep.params.mujoco_ppo \
    --cpanel '{"model_name":"DMBenchCheetahRun-v0", "from_module":"digideep.
↪environment.dmc2gym"}'

python -m digideep.main --params digideep.params.mujoco_ppo \
    --cpanel '{"model_name":"DMBenchCheetahRun-v0", "from_module":"digideep.
↪environment.dmc2gym", "recurrent":True}'
```

Listing 3: Loading a checkpoint to play

```
# Typical loading
python -m digideep.main --play --load-checkpoint "<path-to-checkpoint>"

# Loading a checkpoint using its saved modules (through --save-modules option)
PYTHONPATH="<path-to-session>/modules" python -m digideep.main --play --load-
↪checkpoint "<path-to-checkpoint>"
```

2.2 Playing for Debugging

Listing 4: Command-line arguments

```
$ python -m digideep.environment.play --help
usage: play.py [-h] [--list-include [<pattern>]] [--list-exclude [<pattern>]]
               [--module <module_name>] [--model <model_name>] [--runs <n>]
               [--n-step <n>] [--delay <ms>] [--no-action]

optional arguments:
  -h, --help            show this help message and exit
  --list-include [<pattern>]
                        List by a pattern
  --list-exclude [<pattern>]
                        List by a pattern
  --module <module_name>
                        The name of the module which will register the model
                        in use.
  --model <model_name>  The name of the model to play with random actions.
  --runs <n>            The number of times to run the simulation.
  --n-step <n>          The number of timesteps to run each episode.
  --delay <ms>          The time in milliseconds to delay in each timestep to
```

(continues on next page)

(continued from previous page)

```
                        make simulation slower.
  --no-action           The number of timesteps to run each episode.
```

Listing 5: Running a model with random actions

```
python -m digideep.environment.play --model "Pendulum-v0"
```

Listing 6: Running a model with no actions

```
python -m digideep.environment.play --model "Pendulum-v0" --no-action
```

Listing 7: Running a model from another module (your custom designed environment).

```
python -m digideep.environment.play --model "<model-name>" --module "<module-name>"
```

Listing 8: List registered modules

```
python -m digideep.environment.play --list-include ".*"
python -m digideep.environment.play --list-include ".*Humanoid.*"
python -m digideep.environment.play --list-include ".*Humanoid.*" --list-exclude "DM*"
```

CHAPTER 3

Developer Guide: Big Picture

3.1 The session and runner

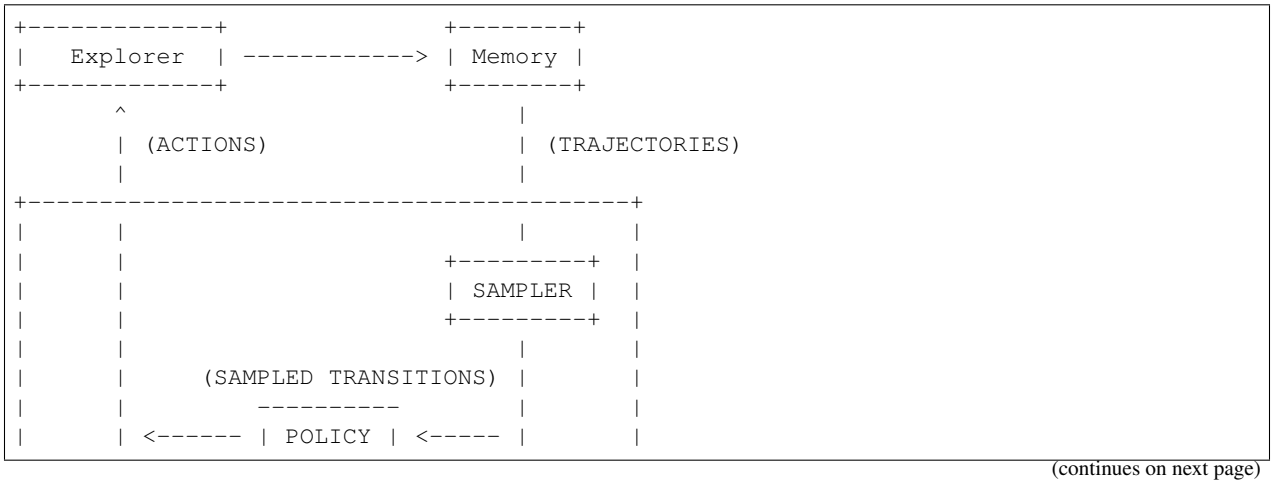
The entrypoint of the program is the `main.py` module. This module, first creates a *Session*.

A *Session* is responsible for command-line arguments, creating a directory for saving the all results related to that session (logs, checkpoints, ...), and initiating the assitive tools, e.g. loggers, monitoring tools, visdom server, etc.

After the *Session* object is created, a *Runner* object is built, either from an existing checkpoint or from the parameters file specified at the command-line. The runner class will run the main loop.

3.2 How does runner work

The *Runner* depends on three main classes: *Explorer*, `Memory`, and `AgentBase`. The connection between these classes is really simple (and is intentionally written to be so), as depicted in the following general graph about reinforcement learning:



(continues on next page)

(continued from previous page)

```
|                  ----------                  |
+-------------------------------------------+
                    AGENT
```

The corresponding (pseudo-)code for the above graph is:

```
do in loop:
    chunk = self.explorer["train"].update()
    self.memory.store(chunk)
    for agent_name in self.agents:
        self.agents[agent_name].update()
```

- *Explorer*: Explorer is responsible for multi-worker environment simulations. It delivers the outputs to the memory in the format of a flattened dictionary (with depth 1). The explorer is tried to be written in its most general manner so it needs least possible modifications for adaptation to new methods.
- Memory: It stores all of the information from the explorer in a dictionary of numpy arrays. The memory is also written in a very general way, so it is usable with most of the methods without modifications.
- *agent*: The agent uses sampler and policy, and is responsible for training the policy and generating actions for simulations in the environment.

CHAPTER 4

Developer Guide: In-Depth Information

In this section, we cover several topics which are essential to understanding how Digideep works.

4.1 Understanding the parameters file

There are two sections in a parameter file. The main section is the `def gen_params(cpanel)` function, which gets the `cpanel` dictionary as its input, and gives the `params` dictionary as the output. The `params` dictionary is the parameter tree of all classes in the project, all in one place. This helps to see the whole structure of the code in one place and have control over them from a centralized location. Moreover, it allows for scripting the parameter relationships, in a more transparent way. Then, there is the `cpanel` dictionary for modifying important parameters from a "control panel". The `cpanel` dictionary may be modified through command-line access:

```
python -m digideep.main ... --cpanel '{"cparam1":"value1", "cparam2":"value2"}'
```

Note: It was possible to implement the parameter file using `json` or `yaml` files. But then it was less intuitive to script the relationships between coupled parameters.

4.2 Understanding the data structure of trajectories

The output of the *`Explorer`*, trajectories, are organized in the form of a dictionary with the following structure:

```
{'/observations':(batch_size, n_steps, ...),
'/masks':(batch_size,n_steps,1),
'/rewards':(batch_size,n_steps,1),
'/infos/<info_key_1>':(batch_size,n_steps,...),
'/infos/<info_key_2>':(batch_size,n_steps,...),
...,
'/agents/<agent_1_name>/actions':(batch_size,n_steps,...),
```

(continues on next page)

(continued from previous page)

```
'/agents/<agent_1_name>/hidden_state':(batch_size,n_steps,...),
'/agents/<agent_1_name>/artifacts/<artifact_1_name>':(batch_size,n_steps,...),
'/agents/<agent_1_name>/artifacts/<artifact_2_name>':(batch_size,n_steps,...),
...,
'/agents/<agent_2_name>/actions':(batch_size,n_steps,...),
'/agents/<agent_2_name>/hidden_state':(batch_size,n_steps,...),
'/agents/<agent_2_name>/artifacts/<artifact_1_name>':(batch_size,n_steps,...),
'/agents/<agent_2_name>/artifacts/<artifact_2_name>':(batch_size,n_steps,...),
...
}
```

Here, `batch_size` is the number of concurrent workers in the *`Explorer`* class, and `n_steps` is the length of each trajectory, i.e. number of timesteps the environment is run.

Note: The names in angle brackets are arbitrary, depending on the agent and environment.

Here's what each entry in the output mean:

- `/observations`: Observations from the environment.
- `/masks`: The `done` flags of the environment. A `mask` value of `0` indicates "finished" episode.
- `/rewards`: The rewards obtained from the environment.
- `/infos/*`: Optional information produced by the environment.
- `/agents/<agent_name>/actions`: Actions took by `<agent_name>`.
- `/agents/<agent_name>/hidden_state`: Hidden_states of `<agent_name>`.
- `/agents/<agent_name>/artifacts/*`: Optional outputs from the agents which includes additional information required for training.

`Memory` will preserve the format of this data structure and store it as it is. `Memory` is basically a queue; new data will replace old data when queue is full.

4.3 Understanding the structure of agents

Digideep supports multiple agents in an environment. Agents are responsible to generate exploratory actions and update their parameters. Agents should inherit `AgentBase`. There are two important components in a typical component: sampler and policy.

Note: The interface of the agent class with the *`Explorer`* is the `action_generator()`. This function is called to generate actions in the environment. The interface of the agent class with the *`Runner`* class is the `update()` class. This function is meant to update the parameters of the agent policy based on collected information from the environment.

As an example of agents, refer to `PPO` or `DDPG`.

4.3.1 Sampler

A sampler samples transitions from the memory to train the policy on. Samplers for different methods share similar parts, thus suggesting to decompose a sampler into smaller units. This obviates developers from some boilerplate

coding. See `digideep.memory.sampler` for some examples.

4.3.2 Policies

Policy is the function inside an agent that generates actions. A policy should inherit from `PolicyBase`. Policies support multi-GPU architectures for inference and architecture. We use `torch.nn.DataParallel` to activate multi-GPU functionalities. Note that using multi-GPUs sometimes does not lead to faster computations, due to larger overheads with respect to gains. It is really problem-dependant.

Every policy should implement the `generate_actions()` function. This function is to be called in the agent's `action_generator()`.

For examples on policies, refer to two available policies in Digideep:

- A stochastic `Policy` for `PPO` agent.
- A deterministic `DDPG` agent.

4.4 Understanding serialization

Digideep is written with serialization in mind from the beginning. The main burden of serialization is on the *Runner* class. It saves both the parameters and states of its sub-components: explorer, memory, and agents. Each of these sub-components are responsible for saving their sub-components states, i.e. in a recursive manner.

Caution: By now, checkpoints only save object states that are necessary for playing the policy, not to resume training.

At each instance of saving two pickle objects are saved, one saving the *Runner*, the other saving the states. "Saving", at its core, is done by using `pickle.dump` for the *Runner* and `torch.save` for the states in the session class. "Loading", uses counterpart functions `pickle.load` and `torch.load` for the *Runner* and states, respectively.

Note: If you are implementing a new method, you should implement your own `state_dict` and `load_state_dict` methods for saving the state of "stateful" objects. Make sure those are called properly during saving and loading.

4.5 Debugging tools

There are some tools commonly used while implementing a reinforcement learning method. We have provided the following assistive tools to help developers debug their codes:

- *digideep.utility.profiling.Profiler*: A lightweight profiling tool. This will help find parts of code that irregularly take more time to complete.
- *digideep.utility.monitoring.Monitor*: A lightweight monitoring tool to keep track of values of variables in training.
- Debugging tools in `digideep.memory.sampler`: There a few sampler units that can be injected into the sampler to inspect shapes, NaN values, and means and standard deviations of a chunk of memory.
- Monitoring CPU/GPU utilization of cores and memory. See *stats* and *runMonitor()*.

4.6 Documentation

We use Sphinx for documentation. If you are not familiar with the syntax, follow the links below:

- Cheat sheet for Google/Numpy style: http://www.sphinx-doc.org/en/master/usage/extensions/napoleon.html
- Basics of reStructuredText: http://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html
- Example Google Style: https://www.sphinx-doc.org/en/1.7/ext/example_google.html

CHAPTER 5

Developer Guide: Implementation Guideline

To implement a new method you need to get a pipeline working as soon as possible. Digideep helps in that manner with developer-friendly source codes, i.e. extensive comments and documentation besides self-descriptive code. The pipeline does not need to train any policies at the beginning.

Digideep is very modular, so that you can use your own implementation for any part instead. However, you are encouraged to fork the source on work on your own copy of the source code for deeper modifications.

5.1 Implementation steps

1. Create a parameter file for your method. You may leave parts that you have not implemented yet blank. Take a look at *digideep.params* for some examples of parameters file or see the descriptions in *Understanding the parameters file*.
2. Create a class for your agent. Inherit from the `AgentBase`.
3. Override `action_generator()` function in your agent's class. Explorer will call this function to generate actions. Follow the expected interface described at `action_generator()`. You can generate random actions but in the correct output shape to get the pipeline done faster.

Tip: Complete your parameters file as you move forward. Run the program early. Try to debug the interface issues as soon as possible.

---

4. In your agent's class, override reset_hidden_state if you are planning to use recurrent policies.
5. Now, the explorer should work fine, and the trajectories may be stored in the memory. Now, it is time to start implementation of your policy.

Note: You should first make sure of correct flow of information through components of the runner, i.e. explorer, memory, and agent, then try to implement the real algorithms. The *Explorer* and `Memory` classes are general classes which can be used with different algorithms.

---

6. To implement your policy, you can inherit from `PolicyBase`.
7. When implementation of policy is done, modify `action_generator()` in your agent to generate actions based on the policy.
8. When policy is done, it's time to implement the sampler for your method. The sampler is typically used at the beginning of the `step()` function of the agent.
9. Implement `step()` function. This is the body of your method. At the same time, `update()` function can be implemented. It is usually just a loop of calls on the `step()` function.
10. At this point, you have successfully finished implementation of your agent. Now it's time to debug. You may use the *Profiler* and *Monitor* tools to inspect the values inside your code and watch the timings.

CHAPTER 6

digideep.pipeline package

6.1 Submodules

6.2 digideep.pipeline.runner module

class digideep.pipeline.runner.**Runner**(*params*)

Bases: object

This class controls the main flow of the program. The main components of the class are:

- explorer: A dictionary containing *Explorer* for the three modes of `train`, `test`, and `eval`. An *Explorer* is a class which handles running simulations concurrently in several environments.
- memory: The component responsible for storing the trajectories generated by the explorer.
- agents: A dictionary containing all agents in the environment.

This class also prints the *Profiler* and *Monitor* information. Also the main serialization burden is on this class. The rest of classes only need to implement the `state_dict` and `load_state_dict` functions for serialization.

> **Caution:** The lines of code for testing while training are commented out.

custom()

enjoy()

This function evaluates the current policy in the environment. It only runs the explorer in a loop.

```
# Do a cycle
while not done:
    # Explore
    explorer["eval"].update()
```

(continues on next page)

(continued from previous page)

```
log()
```

finalize(*save=True*)

instantiate()
This function will instantiate the memory, the explorers, and the agents with their specific parameters.

lazy_connect_signal()

lazy_init()
Initialization of attributes which are not part of the object state. These need lazy initialization due to proper initialization when loading from a checkpoint.

load()
This is a function used by the *start()* function to load the states of internal objects from the checkpoint and update the objects state dicts.

load_memory()

load_state_dict(*state_dict*)
This function will load the states of the internal objects:

- Agents
- Explorers (state of `train` mode would be loaded for `test` and `eval` as well)
- Memory

log()
The log function prints a summary of:

- Frame rate and simulated frames.
- Variables sent to the *Monitor*.
- Profiling information, i.e. registered timing information in the *Profiler*.

monitor_epoch()

on_sigint_received(*signalNumber*, *frame*)

on_sigusr1_received(*signalNumber*, *frame*)

override()

save(*forced=False*)
This is a high-level function for saving both the state of objects and the runner object. It will use helper functions from *Session*.

save_final_checkpoint()

start(*session*)
A function to initialize the objects and load their states (if loading from a checkpoint). This function must be called before using the *train()* and *enjoy()* functions.

If we are starting from scrarch, we will:

- Instantiate all internal components using parameters.

If we are loading from a saved checkpoint, we will:

- Instantiate all internal components using old parameters.
- Load all state dicts.

- (OPTIONAL) Override parameters.

state_dict()
This function will return the states of all internal objects:

- Agents
- Explorer (only the `train` mode)
- Memory

Todo: Memory should be dumped in a separate file, since it can get really large. Moreover, it should be optional.

---

termination_check()

test()

train()
The function that runs the training loop.

See also:

How does runner work

train_cycle()

6.3 digideep.pipeline.session module

class digideep.pipeline.session.**Session**(*root_path*)
Bases: `object`

This class provides the utilities for storing results of a session. It provides a unique path based on a timestamp and creates all sub- folders that are required there. A session directory will have the following contents:

- `session_YYYYMMDDHHMMSS/`:
  - `checkpoints/`: The directory of all stored checkpoints.
  - `modules/`: A copy of all modules that should be saved with the results. This helps to load checkpoints in evolving codes with breaking changes. Use extra modules with `--save-modules` command-line option.
  - `monitor/`: Summary results of each worker environment.
  - `cpanel.json`: A json file including control panel (`cpanel`) parameters in `params` file.
  - `params.yaml`: The parameter tree of the session, i.e. the params variable in `params` file.
  - `report.log`: A log file for Logger class.
  - `visdom.log`: A log file for visdom logs.
  - `__init__.py`: Python `__init__` file to convert the session to a module.

Parameters **root_path** (*str*) – The path to the `digideep` module.

Note: This class also initializes helping tools (e.g. Visdom, Logger, Monitor, etc.) and has helper functions for saving/loading checkpoints.

Tip: The default directory for storing sessions is `/tmp/digideep_sessions`. To change the default directory use the program with cli argument `--session-path <path>`

Todo: Complete the session-as-a-module (SaaM) implementation. Then, `session_YYYYMMDDHHMMSS` should work like an importable module for testing and inference.

Todo: If restoring a session, `visdom.log` should be copied from there and replayed.

play resume loading dry-run session-only | implemented

———————————————————————————————————— | ———— Train 0 0 0 0 0 | 1 Train session barebone 0 0 0 0 1 | 1 Train from a checkpoint 0 1 1 0 0 | 1 Play (policy initialized) 1 0 0 0/1 0 | 1 Play (policy loaded from checkpoint) 1 0 1 0/1 0 | 1

check_if_done()

check_singleton_instance()

createSaaM()
SaaM = Session-as-a-Module This function will make the session act like a python module. The user can then simply import the module for inference.

dump_cpanel(*cpanel*)

dump_params(*params*)

dump_repeal(*repeal*)

finalize()

get_device()

initLogger()
This function sets the logger level and file.

initProlog()

initTensorboard()
Will initialize the SummaryWriter for tensorboard logging.

Link: https://pytorch.org/docs/stable/tensorboard.html

initVarlog()

load_runner()

load_states()

mark_as_done()

parse_arguments()

runMonitor()
This function will load the monitoring tool for CPU and GPU utilization and memory consumption.

save_runner(*runner*, *index*)

save_states(*states*, *index*)

set_device()

update_params(*params*)

digideep.pipeline.session.**check_checkpoint**(*path*, *verbose=False*)

digideep.pipeline.session.**check_session**(*path*, *verbose=False*)

digideep.pipeline.session.**print_verbose**(**args*, *verbose=False*, ***kwargs*)

6.4 Module contents

CHAPTER 7

digideep.params package

7.1 Submodules

7.2 digideep.params.atari_ppo module

See also:

Understanding the parameters file

`digideep.params.atari_ppo.`**`gen_params`**`(`*cpanel*`)`

7.3 digideep.params.classic_ddpg module

This parameter file is designed for continuous action environments. For discrete action environments minor modifications might be required.

See also:

Understanding the parameters file

`digideep.params.classic_ddpg.`**`gen_params`**`(`*cpanel*`)`

7.4 digideep.params.mujoco_ppo module

See also:

Understanding the parameters file

`digideep.params.mujoco_ppo.`**`gen_params`**`(`*cpanel*`)`

7.5 Module contents

CHAPTER 8

digideep.environment package

8.1 Subpackages

8.1.1 digideep.environment.common package

Subpackages

digideep.environment.common.vec_env package

Submodules

digideep.environment.common.vec_env.dummy_vec_env module

class digideep.environment.common.vec_env.dummy_vec_env.**DummyVecEnv**(*env_fns*)
Bases: *digideep.environment.common.vec_env.VecEnv*

VecEnv that does runs multiple environments sequentially, that is, the step and reset commands are send to one environment at a time. Useful when debugging and when num_env == 1 (in the latter case, avoids communication overhead)

get_images()
Return RGB images from each environment

get_rng_state()

load_state_dict(*state_dicts*)

render(*mode='human'*)

reset()
Reset all the environments and return an array of observations, or a dict of observation arrays.

If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

set_rng_state(*states*)

state_dict()

step_async(*actions*)
Tell all the environments to start taking a step with the given actions. Call step_wait() to get the results of the step.

You should not call this if a step_async run is already pending.

step_wait()
Wait for the step taken with step_async().

Returns (obs, rews, dones, infos):

- **obs: an array of observations, or a dict of** arrays of observations.
- rews: an array of rewards
- dones: an array of "episode done" booleans
- infos: a sequence of info objects

digideep.environment.common.vec_env.shmem_vec_env module

OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

class digideep.environment.common.vec_env.shmem_vec_env.**ShmemVecEnv**(*env_fns*, *spaces=None*)

Bases: *digideep.environment.common.vec_env.VecEnv*

Optimized version of SubprocVecEnv that uses shared variables to communicate observations.

close_extras()
Clean up the extra resources, beyond what's in this base class. Only runs when not self.closed.

get_images(*mode='human'*)
Return RGB images from each environment

reset()
Reset all the environments and return an array of observations, or a dict of observation arrays.

If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

step_async(*actions*)
Tell all the environments to start taking a step with the given actions. Call step_wait() to get the results of the step.

You should not call this if a step_async run is already pending.

step_wait()
Wait for the step taken with step_async().

Returns (obs, rews, dones, infos):

- **obs: an array of observations, or a dict of** arrays of observations.
- rews: an array of rewards
- dones: an array of "episode done" booleans
- infos: a sequence of info objects

digideep.environment.common.vec_env.subproc_vec_env module

The MIT License

Copyright (c) 2017 OpenAI (http://openai.com)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

class digideep.environment.common.vec_env.subproc_vec_env.**SubprocVecEnv**(*env_fns*, *spaces=None*)

Bases: *digideep.environment.common.vec_env.VecEnv*

VecEnv that runs multiple environments in parallel in subproceses and communicates with them via pipes. Recommended to use when num_envs > 1 and step() can be a bottleneck.

close_extras()
Clean up the extra resources, beyond what's in this base class. Only runs when not self.closed.

get_images()
Return RGB images from each environment

get_rng_state()

load_state_dict(*state_dicts*)

reset()
Reset all the environments and return an array of observations, or a dict of observation arrays.

If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

set_rng_state(*states*)

state_dict()

step_async(*actions*)
Tell all the environments to start taking a step with the given actions. Call step_wait() to get the results of the step.

You should not call this if a step_async run is already pending.

step_wait()
Wait for the step taken with step_async().

Returns (obs, rews, dones, infos):

- **obs: an array of observations, or a dict of** arrays of observations.
- rews: an array of rewards
- dones: an array of "episode done" booleans
- infos: a sequence of info objects

digideep.environment.common.vec_env.subproc_vec_env.**worker**(*remote*, *parent_remote*, *env_fn_wrapper*)

digideep.environment.common.vec_env.util module

digideep.environment.common.vec_env.util.**copy_obs_dict**(*obs*)
Deep-copy an observation dict.

digideep.environment.common.vec_env.util.**dict_to_obs**(*obs_dict*)
Convert an observation dict into a raw array if the original observation space was not a Dict space.

digideep.environment.common.vec_env.util.**obs_space_info**(*obs_space*)
Get dict-structured information about a gym.Space.

Returns keys: a list of dict keys. shapes: a dict mapping keys to shapes. dtypes: a dict mapping keys to dtypes.

Return type A tuple (keys, shapes, dtypes)

digideep.environment.common.vec_env.util.**obs_to_dict**(*obs*)
Convert an observation into a dict.

digideep.environment.common.vec_env.vec_monitor module

class digideep.environment.common.vec_env.vec_monitor.**VecMonitor**(*venv*, *filename=None*)
Bases: *digideep.environment.common.vec_env.VecEnvWrapper*

reset()
Reset all the environments and return an array of observations, or a dict of observation arrays.

If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

step_wait()
Wait for the step taken with step_async().

Returns (obs, rews, dones, infos):

- **obs: an array of observations, or a dict of** arrays of observations.

- rews: an array of rewards
- dones: an array of "episode done" booleans
- infos: a sequence of info objects

digideep.environment.common.vec_env.vec_video_recorder module

class digideep.environment.common.vec_env.vec_video_recorder.**VecVideoRecorder**(*venv*, *directory*, *record_video_trigg*
video_length=200

Bases: *digideep.environment.common.vec_env.VecEnvWrapper*

Wrap VecEnv to record rendered image as mp4 video.

close()

close_video_recorder()

reset()

Reset all the environments and return an array of observations, or a dict of observation arrays.

If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

start_video_recorder()

step_wait()

Wait for the step taken with step_async().

Returns (obs, rews, dones, infos):

- **obs: an array of observations, or a dict of** arrays of observations.
- rews: an array of rewards
- dones: an array of "episode done" booleans
- infos: a sequence of info objects

Module contents

exception digideep.environment.common.vec_env.**AlreadySteppingError**
Bases: Exception

Raised when an asynchronous step is running while step_async() is called again.

class digideep.environment.common.vec_env.**CloudpickleWrapper**(*x*)
Bases: object

Uses cloudpickle to serialize contents (otherwise multiprocessing tries to use pickle)

exception digideep.environment.common.vec_env.**NotSteppingError**
Bases: Exception

Raised when an asynchronous step is not running but step_wait() is called.

class digideep.environment.common.vec_env.**VecEnv**(*num_envs*, *observation_space*, *action_space*, *spec*, *env_type*)
Bases: abc.ABC

An abstract asynchronous, vectorized environment. Used to batch data from multiple copies of an environment, so that each observation becomes an batch of observations, and expected action is a batch of actions to be applied per-environment.

close()

close_extras()
Clean up the extra resources, beyond what's in this base class. Only runs when not self.closed.

closed = False

get_images()
Return RGB images from each environment

get_viewer()

metadata = {'render.modes': ['human', 'rgb_array']}

render(*mode='human'*)

reset()
Reset all the environments and return an array of observations, or a dict of observation arrays.

If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

step(*actions*)
Step the environments synchronously.

This is available for backwards compatibility.

step_async(*actions*)
Tell all the environments to start taking a step with the given actions. Call step_wait() to get the results of the step.

You should not call this if a step_async run is already pending.

step_wait()
Wait for the step taken with step_async().

Returns (obs, rews, dones, infos):

- **obs: an array of observations, or a dict of** arrays of observations.
- rews: an array of rewards
- dones: an array of "episode done" booleans
- infos: a sequence of info objects

unwrapped

viewer = None

class digideep.environment.common.vec_env.**VecEnvWrapper**(*venv*, *observation_space=None*, *action_space=None*, *spec=None*, *env_type=None*)
Bases: *digideep.environment.common.vec_env.VecEnv*

An environment wrapper that applies to an entire batch of environments at once.

close()

get_images()
Return RGB images from each environment

render(*mode='human'*)

reset()
Reset all the environments and return an array of observations, or a dict of observation arrays.

If step_async is still doing work, that work will be cancelled and step_wait() should not be called until step_async() is invoked again.

step_async(*actions*)
Tell all the environments to start taking a step with the given actions. Call step_wait() to get the results of the step.

You should not call this if a step_async run is already pending.

step_wait()
Wait for the step taken with step_async().

Returns (obs, rews, dones, infos):

- **obs: an array of observations, or a dict of** arrays of observations.
- rews: an array of rewards

- dones: an array of "episode done" booleans
- infos: a sequence of info objects

Submodules

digideep.environment.common.atari_wrappers module

class digideep.environment.common.atari_wrappers.**ClipRewardEnv**(*env*)
Bases: sphinx.ext.autodoc.importer._MockObject

reward(*reward*)
Bin reward to {+1, 0, -1} by its sign.

class digideep.environment.common.atari_wrappers.**EpisodicLifeEnv**(*env*)
Bases: sphinx.ext.autodoc.importer._MockObject

reset(***kwargs*)
Reset only when lives are exhausted. This way all states are still reachable even though lives are episodic, and the learner need not know about any of this behind-the-scenes.

step(*action*)

class digideep.environment.common.atari_wrappers.**FireResetEnv**(*env*)
Bases: sphinx.ext.autodoc.importer._MockObject

reset(***kwargs*)

step(*ac*)

class digideep.environment.common.atari_wrappers.**FrameStack**(*env*, *k*)
Bases: sphinx.ext.autodoc.importer._MockObject

reset()

step(*action*)

class digideep.environment.common.atari_wrappers.**LazyFrames**(*frames*)
Bases: object

class digideep.environment.common.atari_wrappers.**MaxAndSkipEnv**(*env*, *skip=4*)
Bases: sphinx.ext.autodoc.importer._MockObject

reset(***kwargs*)

step(*action*)
Repeat action, sum reward, and max over last observations.

class digideep.environment.common.atari_wrappers.**NoopResetEnv**(*env*, *noop_max=30*)
Bases: sphinx.ext.autodoc.importer._MockObject

reset(***kwargs*)
Do no-op action for a number of steps in [1, noop_max].

step(*ac*)

class digideep.environment.common.atari_wrappers.**ScaledFloatFrame**(*env*)
Bases: sphinx.ext.autodoc.importer._MockObject

observation(*observation*)

class digideep.environment.common.atari_wrappers.**WarpFrame**(*env*, *width=84*, *height=84*, *grayscale=True*)
Bases: sphinx.ext.autodoc.importer._MockObject

observation(*frame*)

digideep.environment.common.atari_wrappers.**make_atari**(*env_id*, *timelimit=True*)

digideep.environment.common.atari_wrappers.**wrap_deepmind**(*env*, *episode_life=True*, *clip_rewards=True*, *frame_stack=False*, *scale=False*)
Configure environment for DeepMind-style Atari.

digideep.environment.common.monitor module

class digideep.environment.common.monitor.**Monitor**(*env*, *filename*, *allow_early_resets=False*, *reset_keywords=()*, *info_keywords=()*)
Bases: sphinx.ext.autodoc.importer._MockObject

EXT = 'monitor.csv'

close()

f = None

get_episode_lengths()

get_episode_rewards()

get_episode_times()

get_total_steps()

reset(***kwargs*)

reset_state()

step(*action*)

update(*ob*, *rew*, *done*, *info*)

digideep.environment.common.monitor.**get_monitor_files**(*dir*)

digideep.environment.common.monitor.**load_results**(*dir*)

digideep.environment.common.running_mean_std module

The MIT License

Copyright (c) 2017 OpenAI (http://openai.com)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

class digideep.environment.common.running_mean_std.**RunningMeanStd**(*epsilon=0.0001*, *shape=()*)

Bases: object

load_state_dict(*state_dict*)

state_dict()

update(*x*)

update_from_moments(*batch_mean*, *batch_var*, *batch_count*)

digideep.environment.common.running_mean_std.**test_runningmeanstd**()

digideep.environment.common.running_mean_std.**update_mean_var_count_from_moments**(*mean*, *var*, *count*, *batch_mean*, *batch_var*, *batch_count*)

digideep.environment.common.tile_images module

`digideep.environment.common.tile_images.`**`tile_images`**(*img_nhwc*)

Tile N images into one big PxQ image (P,Q) are chosen to be as close as possible, and if N is square, then P=Q.

Parameters **`img_nhwc`** – list or array of images, ndim=4 once turned into array n = batch index, h = height, w = width, c = channel

Returns ndarray with ndim=3

Return type bigim_HWc

Module contents

8.1.2 digideep.environment.dmc2gym package

Submodules

digideep.environment.dmc2gym.spec2space module

digideep.environment.dmc2gym.test_dmc2gym module

digideep.environment.dmc2gym.test_pickle module

digideep.environment.dmc2gym.viewer module

digideep.environment.dmc2gym.wrapper module

Module contents

8.2 Submodules

8.3 digideep.environment.data_helpers module

This module provides helper functions to manage data outputs from the *Explorer* class.

See also:

Understanding the data structure of trajectories

digideep.environment.data_helpers.**complete_dict_of_list**(*dic*, *length*)

This function will complete the missing elements of a reference dictionary with similarly-structured `None` values.

Listing 1: Example

```
>>> dic = {'a':[1,2,3,4],
...        'b':[[none,none,none],[none,none,none],[none,none,none],[1,2,3]],
...        'c':[[-1,-2],[-3,-4]]}
>>> # The length of lists under each key is 4 except 'c' which is 2. We have to␣
↪complete that.
>>> complete_dict_of_list(dic, 4)
{'a':[1,2,3,4],
 'b':[[none,none,none],[none,none,none],[none,none,none],[1,2,3]],
 'c':[[-1,-2],[-3,-4],[none,none],[none,none]]}
```

digideep.environment.data_helpers.**convert_time_to_batch_major**(*episode*)

Converts a rollout to have the batch dimension in the major (first) dimension, instead of second dimension.

Parameters **episode** (*dict*) – A trajectory in the form of `{'key1':(num_steps, batch_size,...), 'key2':(num_steps,batch_size,...)}`

Returns A trajectory in the form of `{'key1':(batch_size,num_steps,...), 'key2':(batch_size,num_steps,...)}`

Return type dict

Listing 2: Example

```
>>> episode = {'key1':[[[1],[2]], [[3],[4]], [[5],[6]], [[7],[8]], [[9],[10]]],
               'key2':[[[1,2],[3,4]], [[5,6],[7,8]], [[9,10],[11,12]], [[13,14],
↪[15,16]], [[17,18],[19,20]]]}
>>> convert_time_to_batch_major(episode)
{'key1': array([[[ 1.],
    [ 3.],
    [ 5.],
    [ 7.],
    [ 9.]],

    [[ 2.],
    [ 4.],
    [ 6.],
    [ 8.],
    [10.]]], dtype=float32), 'key2': array([[[ 1.,  2.],
    [ 5.,  6.],
    [ 9., 10.],
    [13., 14.],
    [17., 18.]],

    [[ 3.,  4.],
    [ 7.,  8.],
    [11., 12.],
    [15., 16.],
    [19., 20.]]], dtype=float32)}
```

digideep.environment.data_helpers.**dict_of_lists_to_list_of_dicts**(*dic*, *num*)

Function to convert a dict of lists to a list of dicts. Mainly used to prepare actions to be fed into the `env.step(action)`. `env.step` assumes action to be in the form of a list the same length as the number of workers. It will assign the first action to the first worker and so on.

Parameters

- **dic** (*dict*) – A dictionary with keys being the actions for different agents in the environment.
- **num** (*python:int*) – The number of workers.

Returns A length with its length being same as `num`. Each element in the list would be a dictionary with keys being the agents.

Return type list

Listing 3: Example

```
>>> dic = {'a1':([1,2],[3,4],[5,6]), 'a2':([9],[8],[7])}
>>> num = 3
>>> dict_of_lists_to_list_of_dicts(dic, num)
[{'a1':[1,2], 'a2':[9]}, {'a1':[3,4], 'a2':[8]}, {'a1':[5,6], 'a2':[7]}]
```

Caution: This only works for 1-level dicts, not for nested dictionaries.

digideep.environment.data_helpers.**extract_keywise**(*dic*, *key*)

This function will extract a key from all entries in a dictionary. Key should be first-level key.

Parameters

- **dic** (*dict*) – The input dictionary containing a dict of dictionaries.
- **key** – The key name to be extracted.

Returns The result dictionary

Return type dict

Listing 4: Example

```
>>> dic = {'agent1':{'a':[1,2],'b':{'c':2,'d':4}}, 'agent2':{'a':[3,4],'b':{'c':9,
↪'d':7}}}
>>> key = 'a'
>>> extract_keywise(dic, key)
{'agent1':[1,2], 'agent2':[3,4]}
```

digideep.environment.data_helpers.**flatten_dict**(*dic*, *sep='/'*, *prefix=''*)

We flatten a nested dictionary into a 1-level dictionary. In the new dictionary keys are combinations of previous keys, separated by the `sep`. We follow unix-style file system naming.

Listing 5: Example

```
>>> Dict = {"a":1, "b":{"c":1, "d":{"e":2, "f":3}}}
>>> flatten_dict(Dict)
{"/a":1, "/b/c":1, "/b/d/e":2, "/b/d/f":3}
```

digideep.environment.data_helpers.**flattened_dict_of_lists_to_dict_of_numpy**(*dic*)

digideep.environment.data_helpers.**join_keys**(*key1*, *key2*, *sep='/'*)

Parameters

- **key1** (*str*) – The first key in unix-style file system path.
- **key1** – The second key in unix-style file system path.
- **sep** (*str*) – The separator to be used.

Listing 6: Example

```
>>> join_keys('/agent1','artifcats')
'/agent1/artifacts'
```

digideep.environment.data_helpers.**list_of_dicts_to_flattened_dict_of_lists**(*List*, *length*)

Function to convert a list of (nested) dicts to a flattened dict of lists. See the example below.

Parameters

- **List** (*list*) – A list of dictionaries. Each element in the list is a single sample data produced from the environment.
- **length** (*python:int*) – The length of time sequence. It is used to complete the data entries which were lacking from some data samples.

Returns A dictionary whose keys are flattened similar to Unix-style file system naming.

Return type dict

Listing 7: Example

```
>>> List = [{'a':{'f':[1,2], 'g':[7,8]}, 'b':[-1,-2], 'info':[10,20]},
            {'a':{'f':[3,4], 'g':[9,8]}, 'b':[-3,-4], 'step':[80,90]}]
>>> Length = 2
>>> list_of_dicts_to_flattened_dict_of_lists(List, Length)
{'/a/f':[[1,2],[3,4]],
'/a/g':[[7,8],[9,8]],
    'b':[[-1,-2],[-3,-4]],
    '/info':[[10,20],[none,none]],
    '/step':[[none,none],[80,90]]}
```

Listing 8: Example

```
# Intermediate result, before doing ``complete_dict_of_list``:
{'/a/f':[[1,2],[3,4]],
'/a/g':[[7,8],[9,8]],
'b':[[-1,-2],[-3,-4]],
'/info':[[10,20]],
'/step':[[none,none],[80,90]]}
# Final result, after doing ``complete_dict_of_list`` ('/info' will become␣
↪complete in length):
{'/a/f':[[1,2],[3,4]],
'/a/g':[[7,8],[9,8]],
'b':[[-1,-2],[-3,-4]],
'/info':[[10,20],[none,none]],
'/step':[[none,none],[80,90]]}
```

digideep.environment.data_helpers.**nonify**(*element*)

This function creates an output with all elements being `None`. The structure of the resulting element is exactly the structure of the input `element`. The `element` cannot contain dicts. The only accepted types are `tuple`, `list`, and `np.ndarray`. It can contain nested lists and tuples, however.

Listing 9: Example

```
>>> Input = [(1,2,3), (1,2,4,5,[-1,-2])]
>>> nonify(Input)
[(none,none,none), (none,none,none,none,[none,none])]
```

digideep.environment.data_helpers.**unflatten_dict**(*dic*, *sep='/'*)

Unflattens a flattened dictionary into a nested dictionary.

Listing 10: Example

```
>>> Dict = {"/a":1, "/b/c":1, "/b/d/e":2, "/b/d/f":3}
>>> unflatten_dict(Dict)
{"a":1, "b":{"c":1, "d":{"e":2, "f":3}}}
```

digideep.environment.data_helpers.**update_dict_of_lists**(*dic*, *item*, *index=0*)

This function updates a dictionary with a new item.

Listing 11: Example

```
>>> dic = {'a':[1,2,3], 'c':[[-1,-2],[-3,-4]]}
>>> item = {'a':4, 'b':[1,2,3]}
>>> index = 3
>>> update_dict_of_lists(dic, item, index)
{'a':[1,2,3,4],
 'b':[[none,none,none],[none,none,none],[none,none,none],[1,2,3]],
 'c':[[-1,-2],[-3,-4]]}
```

Note: `c` in the above example is not "complete" yet! The function *`complete_dict_of_list()`* will complete the keys which need to be completed!

Caution: This function does not support nested dictionaries.

8.4 digideep.environment.explorer module

class digideep.environment.explorer.**Explorer**(*session*, *agents=None*, ***params*)

Bases: `object`

A class which runs environments in parallel and returns the result trajectories in a unified structure. It support multi-agents in an environment.

Note: The entrypoint of this class is the *`update()`* function, in which the *`step()`* function will be called for `n_steps` times. In the *`step()`* function, the *`prestep()`* function is called first to get the actions from the agents. Then the `env.step` function is called to execute those actions in the environments. After the loop is done in the *`update()`*, we do another *`prestep()`* to save the `observations`/`actions` of the last step. This indicates the final action that the agent would take without actually executing that. This information will be useful in some algorithms.

Parameters

- **session** (*Session*) – The running session object.
- **agents** (*dict*) – A dictionary of the agents and their corresponding agent objects.
- **mode** (*str*) – The mode of the Explorer, which is any of the three: `train|test|eval`
- **env** (`env`) – The parameters of the environment.
- **do_reset** (*bool*) – A flag indicating whether to reset the environment at the update start.
- **final_action** (*bool*) – A flag indicating whether in the final call of *prestep()* the action should also be generated or not.
- **num_workers** (*python:int*) – Number of workers to work in parallel.
- **deterministic** (*bool*) – Whether to choose the optimial action or to mix some noise with the action (i.e. for exploration).
- **n_steps** (*python:int*) – Number of steps to take in the *update()*.
- **render** (*bool*) – A flag used to indicate whether environment should be rendered at each step.
- **render_delay** (*python:float*) – The amount of seconds to wait after calling `env.render`. Used when environment is too fast for visualization, typically in `eval` mode.
- **seed** (*python:int*) – The environment seed.

Variables

- **steps** (*python:int*) – Number of times the *step()* function is called.
- **n_episode** (*python:int*) – Number of episodes (a full round of simulation) generated so far.
- **timesteps** (*python:int*) – Number of total timesteps of experience generated so far.
- **was_reset** (*bool*) – A flag indicating whether the Explorer has been just reset or not.
- **observations** – A tracker of environment observations used to produce the actions for the next step.
- **masks** – A tracker of environment `done` flag indicating the start of a new episode.
- **hidden_states** – A tracker of hidden_states of the agents for producing the next step action in recurrent policies.

Caution: Use `do_reset` with caution; only when you know what the consequences are. Generally there are few oportunities when this flag needs to be true.

Tip: This class is partially serializable. It only saves the state of environment wrappers and not the environment per se.

See also:

Understanding the data structure of trajectories

close()
It closes all environments.

load_state_dict(*state_dict*)

monitor_n_episode()

monitor_timesteps()

prestep(*final_step=False*)
Function to produce actions for all of the agents. This function does not execute the actions in the environment.

Parameters **final_step** (*bool*) – A flag indicating whether this is the last call of this function.

Returns The pre-transition dictionary containing observations, masks, and agents informations. The format is like: `{"observations":..., "masks":..., "agents":...}`

Return type dict

report_rewards(*infos*)
This function will extract episode information from infos and will send them to *Monitor* class.

reset()
Will reset the Explorer and all of its states. Will set `was_reset` to `True` to prevent immediate resets.

state_dict()

step()
Function that runs the `prestep` and the actual `env.step` functions. It will also manipulate the transition data to be in appropriate format.

Returns The full transition information, including the pre-transition (actions, last observations, etc) and the results of executing actions on the environments, i.e. rewards and infos. The format is like: `{"observations":..., "masks":..., "rewards":..., "infos":..., "agents":...}`

Return type dict

See also:

Understanding the data structure of trajectories

update()
Runs *step()* for `n_steps` times.

Returns A dictionary of unix-stype file system keys including all information generated by the simulation.

Return type dict

See also:

Understanding the data structure of trajectories

8.5 digideep.environment.make_environment module

This module is inspired by pytorch-a2c-ppo-acktr.

class digideep.environment.make_environment.**MakeEnvironment**(*session*, *mode*, *seed*, ***params*)

Bases: `object`

This class will make the environment. It will apply the wrappers to the environments as well.

Tip: Except *Monitor* environment, no environment will be applied on the environment unless explicitly specified.

create_envs(*num_workers=1*, *force_no_monitor=False*, *extra_env_kwargs={}*)

get_config()
This function will generate a dict of interesting specifications of the environment.

Note: Observation and action can be nested spaces.Dict.

make_env(*rank*, *force_no_monitor=False*, *extra_env_kwargs={}*)

registered = False

run_wrapper_stack(*env*, *stack*)
Apply a series of wrappers.

digideep.environment.make_environment.**space2config**(*S*)
Function to convert space's characteristics into a config-space dict.

8.6 digideep.environment.play module

8.7 digideep.environment.wrappers module

8.8 Module contents

CHAPTER 9

digideep.memory package

CHAPTER 10

digideep.agent package

10.1 Submodules

10.2 digideep.agent.base module

10.3 digideep.agent.ddpg module

10.4 digideep.agent.noises module

This module is dedicated to noise models used in other methods.

Each noise class should implement the `__call__` method. See the examples *EGreedyNoise* and *OrnsteinUhlenbeckNoise*.

class digideep.agent.noises.**EGreedyNoise**(***params*)

Bases: object

This class implements simple e-greedy noise. The noise is sampled from uniform distribution.

Parameters

- **std** (*python:float*) – Standard deviation of the noise.
- **e** (*python:float*) – The probability of choosing a noisy action.
- **lim** (*python:float*) – Boundary of the noise (noise will be clipped beyond this value.)

Note: This class is not dependant on its history.

load_state_dict(*state_dict*)

reset()

state_dict()

class digideep.agent.noises.**OrnsteinUhlenbeckNoise**(***params*)
Bases: object

An implementation of the Ornstein-Uhlenbeck noise.

The noise model is $dx_t = heta(\mu - x_t)\,dt + \sigma\,dW_t$.

Parameters

- **mu** – Parameter μ which indicates the final value that x will converge to.
- **theta** – Parameter :math:‘ heta‘.
- **sigma** – Parameter σ which is the std of the additional normal noise.
- **lim** – The action limit, which can be a `np.array` for a vector of actions.

Note: This class is state serializable.

---

load_state_dict(*state_dict*)

reset(*action*)

state_dict()

10.5 digideep.agent.ppo module

10.6 Module contents

CHAPTER 11

digideep.policy package

11.1 Subpackages

11.1.1 digideep.policy.deterministic package

Submodules

digideep.policy.deterministic.policy module

Module contents

11.1.2 digideep.policy.stochastic package

Submodules

digideep.policy.stochastic.blocks module

digideep.policy.stochastic.common module

digideep.policy.stochastic.distributions module

digideep.policy.stochastic.policy module

Module contents

11.2 Submodules

11.3 digideep.policy.base module

11.4 Module contents

CHAPTER 12

digideep.utility package

12.1 Subpackages

12.1.1 digideep.utility.visdom_engine package

Submodules

digideep.utility.visdom_engine.Instance module

class digideep.utility.visdom_engine.Instance.**VisdomInstance**(*port=8097*, *log_to_filename=None*, *replay=True*)

Bases: object

This class is a singleton for getting an instance of Visdom client. It also replays all the logs at the loading time.

Session is responsible for initializing the log_file and replaying the old log.

Parameters

- **port** (*python:int*) – The port number of the running Visdom server.
- **log_to_filename** (*str*) – The log file for the Visdom server.
- **replay** (*bool, False*) – Whether to replay from existing Visdom log files in the path. Use with care if the log file is very big.

static getVisdomInstance(***kwargs*)

Static access method.

digideep.utility.visdom_engine.WebServer module

class digideep.utility.visdom_engine.WebServer.**VisdomWebServer**(*port=8097*, *enable_login=False*, *username='visdom'*, *password='visdom'*, *cookie_secret='visdom@d1c11598d2fb'*)

Bases: object

This class runs a Visdom Server.

Parameters

- **port** (*python:int*) – Port for server to run on.
- **enable_login** (*bool*) – Whether to activate login screen for the server.
- **username** (*str*) – The username for login.
- **password** (*str*) – The password for login. A hashed version of the password will be stored in the Visdom settings.
- **cookie_secret** (*str*) – A unique string to be used as a cookie for the server.

run()
Method that runs forever

digideep.utility.visdom_engine.Wrapper module

This module is highly inspired by: https://github.com/pytorch/tnt

class digideep.utility.visdom_engine.Wrapper.**VisdomWrapper**(*command*, *win*, ***kwargs*)

Bases: object

This class does not need to be serializable.

Parameters

- **command** – The visdom command.
- **win** – The window name.
- **kwargs** – The dictionary of keyword arguments. May include opts and env.

Note: If you want to be consistent between different runs, you must assign 'win' as input.

Example

```
>>> v = VisdomWrapper('line', win='TestLoss', opts={'title':'TestLoss'}, X=np.
↪array([1]), Y=np.array([4]))
```

get_env()

get_win()

run(***kwargs*)

class digideep.utility.visdom_engine.Wrapper.**VisdomWrapperPlot**(*plot_type*, *win*, ***kwargs*)

Bases: *digideep.utility.visdom_engine.Wrapper.VisdomWrapper*

In the append function, user should provide X=np.array(...), Y=np.array(...)

append(***kwargs*)

Module contents

12.2 Submodules

12.3 digideep.utility.filter module

class digideep.utility.filter.**MovingAverage**(*size=1*, *window_size=10*)

Bases: object

An implementation of moving average. It has an internal queue of the values.

Parameters

- **size** (*python:int*) – The length of the value vector.
- **window_size** (*python:int*) – The window size for calculation of the moving average.

append(*item*)

data

max

mean

median

min

std

12.4 digideep.utility.logging module

class digideep.utility.logging.**Logger**
Bases: object

This is a helper class which is intended to simplify logging in a single file from different modules in a package. The Logger class uses a singleton[1] pattern.

It also provides multi-level logging each in a specific style. The levels are DEBUG, INFO, WARN, ERROR, FATAL.

Listing 1: Example

```
logger.set_log_level(2)
logger.info('This is a test of type INFO.')    # Will not be shown
logger.warn('This is a test of type WARN.')    # Will be shown
logger.fatal('This is a test of type FATAL.') # Will be shown

logger.set_log_level(3)
logger.info('This is a test of type INFO.')    # Will not be shown
logger.warn('This is a test of type WARN.')    # Will not be shown
logger.fatal('This is a test of type FATAL.') # Will be shown

logger.set_logfile('path_to_the_log_file')
# ... All logs will be stored in the specified file from now on.
#     They will be shown on the output as well.
```

debug(**args*, *sep=' '*, *end='\n'*, *flush=False*)

error(**args*, *sep=' '*, *end='\n'*, *flush=False*)

fatal(**args*, *sep=' '*, *end='\n'*, *flush=False*)

static getInstance()
Static access method.

info(**args*, *sep=' '*, *end='\n'*, *flush=False*)

set_log_level(*log_level*)

set_logfile(*filename*)

warn(**args*, *sep=' '*, *end='\n'*, *flush=False*)

12.5 digideep.utility.monitoring module

class digideep.utility.monitoring.**Monitor**
Bases: object

A very simple and lightweight implementation for a global monitoring tool. This class keeps track of a variable's mean, standard deviation, minimum, maximum, and sum in a recursive manner.

[1] https://gist.github.com/pazdera/1098129

```
>>> monitor.reset()
>>> for i in range(1000):
...   monitor('loop index', i)
...
>>> print(monitor)
>> loop index [1000x] = 499.5 (+-577.639 %95) in range{0 < 999}
```

Todo: Provide batched monitoring of variables.

Note: This class does not implement moving average. For a moving average implementation refer to *MovingAverage*.

discard_key(*key*)

dump()

get_meta_key(*key*)

pack_data()

pack_keys(*keys*)

reset()

set_meta_key(*key*, *value*)

set_output_file(*path*)

class digideep.utility.monitoring.**WindowValue**(*value*, *window*)

Bases: object

append(*value*)

get_max()

get_min()

get_num()

get_std()

get_sum()

get_win()

12.6 digideep.utility.plotting module

class digideep.utility.plotting.**Plotter**(**params*)

Bases: object

append(*y*, *x=None*)

12.7 digideep.utility.profiling module

class digideep.utility.profiling.**KeepTime**(*name*)

Bases: object

add_name()

get_current_level()

get_full_path()

get_level()

pop_name()

set_level()

class digideep.utility.profiling.**Profiler**

Bases: object

This class provides a very simple yet light implementation of function profiling. It is very easy to use:

```
>>> profiler.reset()
>>> profiler.start("loop")
>>> for i in range(100000):
...   print(i)
...
>>> profiler.lapse("loop")
>>> print(profiler)
>> loop [1x, 27.1s]
```

Alternatively, you may use `profiler` with *KeepTime*:

```
>>> with KeepTime("loop2"):
...   for i in range(100000):
...     print(i)
...
>>> print(profiler)
>> loop2 [1x, 0.0s]
```

Note: The number of callings to *start()* and *lapse()* should be the same.

dump(*meta={}*)

get_keys()

get_occurence(*name*)

get_time_average(*name*)

get_time_overall(*name*)

lapse(*name*)

reset()

set_output_file(*path*)

start(*name*)

12.8 digideep.utility.stats module

12.9 digideep.utility.timer module

class digideep.utility.timer.**Timer**(*task*, *interval=1.0*)
Bases: threading.Thread

Thread that executes a task every N seconds

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

setInterval(*interval*)
Set the number of seconds we sleep between executing our task

shutdown()
Stop this thread

task_exec()
The task done by this thread - override in subclasses

12.10 digideep.utility.toolbox module

digideep.utility.toolbox.**count_parameters**(*model*)
Counts the number of parameters in a PyTorch model.

digideep.utility.toolbox.**dump_dict_as_json**(*filename*, *dic*, *sort_keys=False*)
This function dumps a python dictionary in `json` format to a file.

Parameters

- **filename** (*path*) – The address to the file.
- **dic** (*dict*) – The dictionary to be dumped in json format. It should be json-serializable.
- **sort_keys** (*bool, False*) – Will sort the dictionary by its keys before dumping to the file.

digideep.utility.toolbox.**dump_dict_as_yaml**(*filename*, *dic*)

digideep.utility.toolbox.**get_class**(*addr*)
Return a instance of a class by using only its name.

Parameters **addr** (*str*) – The name of the class/function which should be in the format `MODULENAME[.SUBMODULE1[.SUBMODULE2[...]]].CLASSNAME`

digideep.utility.toolbox.**get_module**(*addr*)
Return a instance of a module by using only its name.

Parameters **addr** (*str*) – The name of the module which should be in the format `MODULENAME[.SUBMODULE1[.SUBMODULE2[...]]]`

digideep.utility.toolbox.**get_rng_state**()

digideep.utility.toolbox.**load_json_as_dict**(*filename*)

digideep.utility.toolbox.**load_yaml_as_dict**(*filename*)

digideep.utility.toolbox.**seed_all**(*seed*, *cuda_deterministic=False*)

digideep.utility.toolbox.**set_rng_state**(*states*)

digideep.utility.toolbox.**strict_update**(*dict_target*, *dict_source*)

12.11 Module contents

CHAPTER 13

Indices and tables

- genindex
- modindex

Python Module Index

d

Index

E

F

G

I

J

K

L

M

N

O

P

R

S

T

U

V

W