baconian Documentation

Release 0.2.6

Linsen Dong

May 22, 2020

Installation

CHAPTER 1

Installation Guide

Baconian is easy to install. We offer the pip requirement file to install the required packages. Make sure your machine has python 3.5, 3.6, or 3.7 with Ubuntu 16.04, 18.04 (recommend python 3.5 and ubuntu 16.04).

1. We recommend you to use anaconda to manage your package and environment, since installing the required packages may overwrite some of already installed packages with a different version.

```
source activate your_env
```

2. Install by pip or clone the source code and install the packages with requirement file:

```
// pip install:
pip install baconian
// source code install
git clone git@github.com:cap-ntu/baconian-project.git baconian
cd ./baconian
pip install pip -U
pip install tensorflow==1.15.2 // or pip install tensorflow-gpu==1.15.2
pip install -e .
```

Then you are free to go. You can either use the Baconian as a third party package and import it into your own project, or directly modify it.

After you finish the above installation, you are able to run the following environments of Gym:

- algorithmic
- toy_text
- classic_control

3. Support for Gym full environments

If you want to use the full environments of gym, please refer to gym to obtain the license and library.

Mostly you will need to install requirements for mujoco environments and box-2d environments.

4. Support for DeepMind Control Suit

For DeepMind control suit, you should install it by:

```
pip install git+git://github.com/deepmind/dm_control.git
```

DeepMind control suit also relies on the mujoco engine which is the same as the mujoco-py environments in gym. They provide similar tasks with slightly differences.

And for the different default mujoco key and mujoco binaries for mujoco-py and DeepMind control suit, please follow the setting of mujoco-py and we will take care of the setting for DeepMind Control Suit at runtime.

5. We have implemented many examples, you may try them first at *Examples*
6. If you prefer to conduct a new experiments by yourself, you can follow the tutorial here *How to implement*

CHAPTER 2

Step by step guide to run a RL experiment

This is a step by step guide of how to compose an model-based RL experiments in Baconian, we will take the example of Dyna algorithm, which is a very typical model-based RL architecture proposed by Sutton in 1990.

In this method, we need the environment and the task we aim to solve, a model-free method, an approximator as dynamics for real world environment. We also need to specific the hyper-parameters in Dyna architecture, e.g., how many samples we should get from real environment to update the dynamics model, and how many samples from real environment and dynamics model to serve the training of the model-free algorithms.

For more information on the modules used in the following codes, please see *Best Practices* and *API references*

Note: Complete codes can be found at *here*

2.1 Create the tasks and DDPG algorithms

Create environment Pendulum-v0, and the DDPG algorithms to solve the task

```
def task_fn():
    # create the gym environment by make function
    env = make('Pendulum-v0')
    # give your experiment a name which is used to generate the log path etc.
    name = 'demo_exp'
    # construct the environment specification
    env_spec = EnvSpec(obs_space=env.observation_space,
                       action_space=env.action_space)
    # construct the neural network to approximate q function of DDPG
    mlp_q = MLPQValueFunction(env_spec=env_spec,
                              name_scope=name + '_mlp_q',
                              name=name + '_mlp_q',
                              mlp_config=[
                                  {
                                      "ACT": "RELU",
```

(continues on next page)

(continued from previous page)

```
                                    "B_INIT_VALUE": 0.0,
                                    "NAME": "1",
                                    "N_UNITS": 16,
                                    "TYPE": "DENSE",
                                    "W_NORMAL_STDDEV": 0.03
                                },
                                {
                                    "ACT": "LINEAR",
                                    "B_INIT_VALUE": 0.0,
                                    "NAME": "OUPTUT",
                                    "N_UNITS": 1,
                                    "TYPE": "DENSE",
                                    "W_NORMAL_STDDEV": 0.03
                                }
                            ])
    # construct the neural network to approximate policy for DDPG
    policy = DeterministicMLPPolicy(env_spec=env_spec,
                                    name_scope=name + '_mlp_policy',
                                    name=name + '_mlp_policy',
                                    mlp_config=[
                                        {
                                            "ACT": "RELU",
                                            "B_INIT_VALUE": 0.0,
                                            "NAME": "1",
                                            "N_UNITS": 16,
                                            "TYPE": "DENSE",
                                            "W_NORMAL_STDDEV": 0.03
                                        },
                                        {
                                            "ACT": "LINEAR",
                                            "B_INIT_VALUE": 0.0,
                                            "NAME": "OUPTUT",
                                            "N_UNITS": env_spec.flat_action_dim,
                                            "TYPE": "DENSE",
                                            "W_NORMAL_STDDEV": 0.03
                                        }
                                    ],
                                    reuse=False)
    # construct the DDPG algorithms
    ddpg = DDPG(
        env_spec=env_spec,
        config_or_config_dict={
            "REPLAY_BUFFER_SIZE": 10000,
            "GAMMA": 0.999,
            "CRITIC_LEARNING_RATE": 0.001,
            "ACTOR_LEARNING_RATE": 0.001,
            "DECAY": 0.5,
            "BATCH_SIZE": 50,
            "TRAIN_ITERATION": 1,
            "critic_clip_norm": 0.1,
            "actor_clip_norm": 0.1,
        },
        value_func=mlp_q,
        policy=policy,
        name=name + '_ddpg',
        replay_buffer=None
    )
```

2.2 Create a global dynamics model with MLP network

```
    # construct a neural network based global dynamics model to approximate the state transition of environment
    mlp_dyna = ContinuousMLPGlobalDynamicsModel(
        env_spec=env_spec,
        name_scope=name + '_mlp_dyna',
        name=name + '_mlp_dyna',
        learning_rate=0.01,
        state_input_scaler=RunningStandardScaler(dims=env_spec.flat_obs_dim),
        action_input_scaler=RunningStandardScaler(dims=env_spec.flat_action_dim),
        output_delta_state_scaler=RunningStandardScaler(dims=env_spec.flat_obs_dim),
        mlp_config=[
            {
                "ACT": "RELU",
                "B_INIT_VALUE": 0.0,
                "NAME": "1",
                "L1_NORM": 0.0,
                "L2_NORM": 0.0,
                "N_UNITS": 16,
                "TYPE": "DENSE",
                "W_NORMAL_STDDEV": 0.03
            },
            {
                "ACT": "LINEAR",
                "B_INIT_VALUE": 0.0,
                "NAME": "OUPTUT",
                "L1_NORM": 0.0,
                "L2_NORM": 0.0,
                "N_UNITS": env_spec.flat_obs_dim,
                "TYPE": "DENSE",
                "W_NORMAL_STDDEV": 0.03
            }
        ])
```

2.3 Create the Dyna architecture as algorithms

Create the Dyna algorithms by passing the ddpg and dynamics model, and wrap by the agent

```
    # finally, construct the Dyna algorithms with a model free algorithm DDGP, and a NN model.
    algo = Dyna(env_spec=env_spec,
                name=name + '_dyna_algo',
                model_free_algo=ddpg,
                dynamics_model=mlp_dyna,
                config_or_config_dict=dict(
                    dynamics_model_train_iter=10,
                    model_free_algo_train_iter=10
                ))
    # To make the NN based dynamics model a proper environment so be a sampling source for DDPG, reward function and
    # terminal function need to be set.

    # For examples only, we use random reward function and terminal function with fixed episode length.
```

(continues on next page)

(continued from previous page)

```
    algo.set_terminal_reward_function_for_dynamics_env(
        terminal_func=FixedEpisodeLengthTerminalFunc(max_step_length=env.unwrapped._
↪max_episode_steps,
                                                     step_count_fn=algo.dynamics_env.
↪total_step_count_fn),
        reward_func=RandomRewardFunc())
    # construct agent with additional exploration strategy if needed.
    agent = Agent(env=env, env_spec=env_spec,
                  algo=algo,
                  algo_saving_scheduler=PeriodicalEventSchedule(
                      t_fn=lambda: get_global_status_collect()('TOTAL_AGENT_TRAIN_
↪SAMPLE_COUNT'),
                      trigger_every_step=20,
                      after_t=10),
                  name=name + '_agent',
                  exploration_strategy=EpsilonGreedy(action_space=env_spec.action_
↪space,
                                                     init_random_prob=0.5))
```

2.4 Configure the workflow and experiments object

Create the dyna-like workflow, and the experiments object, and run the experiment within your *task_fn*

```
    # construct the training flow, called Dyna flow. It defines how the training␣
↪proceed, and the terminal condition
    flow = create_dyna_flow(
        train_algo_func=(agent.train, (), dict(state='state_agent_training')),
        train_algo_from_synthesized_data_func=(agent.train, (), dict(state='state_
↪agent_training')),
        train_dynamics_func=(agent.train, (), dict(state='state_dynamics_training')),
        test_algo_func=(agent.test, (), dict(sample_count=1)),
        test_dynamics_func=(agent.algo.test_dynamics, (), dict(sample_count=10,␣
↪env=env)),
        sample_from_real_env_func=(agent.sample, (), dict(sample_count=10,
                                                          env=agent.env,
                                                          store_flag=True)),
        sample_from_dynamics_env_func=(agent.sample, (), dict(sample_count=10,
                                                              env=agent.algo.dynamics_
↪env,
                                                              store_flag=True)),
        train_algo_every_real_sample_count_by_data_from_real_env=40,
        train_algo_every_real_sample_count_by_data_from_dynamics_env=40,
        test_algo_every_real_sample_count=40,
        test_dynamics_every_real_sample_count=40,
        train_dynamics_ever_real_sample_count=20,
        start_train_algo_after_sample_count=1,
        start_train_dynamics_after_sample_count=1,
        start_test_algo_after_sample_count=1,
        start_test_dynamics_after_sample_count=1,
        warm_up_dynamics_samples=1
    )
    # construct the experiment
    experiment = Experiment(
        tuner=None,
```

(continues on next page)

(continued from previous page)

```
        env=env,
        agent=agent,
        flow=flow,
        name=name + '_exp'
    )
    # run!
    experiment.run()


from baconian.core.experiment_runner import *
```

Note: Don't confuse the workflow with the Dyna algorithm itself. The flow only specifics how the algorithms interact with environments, and how to update and evaluate the ddpg (model-free method) and dynamics model.

2.5 Set the global configuration and launch the experiment

Set some global config if needed and wrap all the above task into a function *task_fn*, and pass into experiment runner.

```
# set some global configuration here

# set DEFAULT_EXPERIMENT_END_POINT to indicate when to stop the experiment.
# one usually used is the TOTAL_AGENT_TRAIN_SAMPLE_COUNT, i.e., how many samples/
↪timesteps are used for training
GlobalConfig().set('DEFAULT_EXPERIMENT_END_POINT', dict(TOTAL_AGENT_TRAIN_SAMPLE_
↪COUNT=200))

# set the logging path to write log and save model checkpoints.
GlobalConfig().set('DEFAULT_LOG_PATH', './log_path')

# feed the task into a exp runner.
single_exp_runner(task_fn, del_if_log_path_existed=True)
```

2.6 Results logging/ Visualization

Please refer to *Logging and Visualization*.

CHAPTER 3

Best practice and core concepts of Baconian

Here we introduce the core ideas and features about Baconian, to make sure you utilize the code correctly. As for the detailed usage of different algorithms, dynamics, please refer to the *API* page

3.1 Put your task into a function

Here we introduce the basic usage of Baconian, and introduce how it can help you to set up the model-based RL experiments.

First of all, whenever you want to run some algorithms, or any codes within the baconian, simply you need to wrap your code into a function, and pass this function to `single_exp_runner` or `duplicate_exp_runner`. In this method, Baconian will do some internal initialization of logging, experiment set-up etc.

`single_exp_runner` will run your function for once. As for `duplicate_exp_runner`, it is designed for running multiple experiments in a row, because in RL experiments, we usually run the experiment with a certain set of parameters but with different seeds to get a more stable results. So use `duplicate_exp_runner` can easily help you to achieve this, and the log file will be stored into sub-directory under your home log directory respectively.

Specifically, you can do it by:

```
from baconian.core.experiment_runner import single_exp_runner, duplicate_exp_runner
# Define your function first.
def your_function():
    a = 1
    b = 2
    print(a + b)
# Then pass the function object to single_exp_runner, then it will set up everything␣
↪and run your code.
single_exp_runner(your_function)
# Or call duplicate_exp_runner to run multiple experiments in a row. 10 is the number␣
↪of experiments:
duplicate_exp_runner(10, your_function)
```

3.2 Global Configuration

The global configuration offer the setting including default log path, log level, and some other system related default configuration. We implement the global configuration module with singleton method, and you can utilize it by following examples:

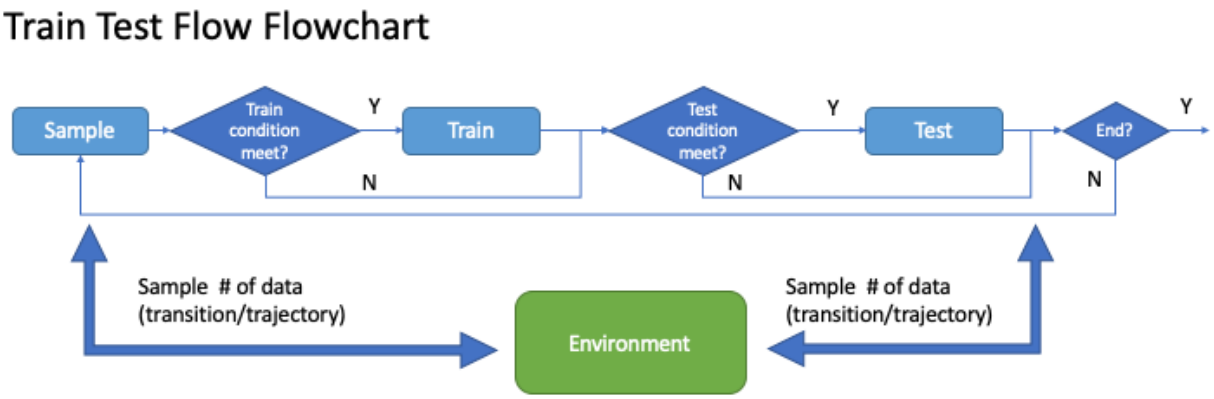
```
from baconian.config.global_config import GlobalConfig
from baconian.core.experiment_runner import single_exp_runner, duplicate_exp_runner
def your_function():
    a = 1;
    b = 2;
    print(a + b)
# Use GlobalConfig() to access the instance of GlobalConfig
# anywhere your want, and set the log path by yourself
# First argument is key you want to set, e.g., DEFAULT_LOG_PATH
# Second argument is the value.
GlobalConfig().set('DEFAULT_LOG_PATH', './log_path')
single_exp_runner(task_fn, del_if_log_path_existed=True)
```

During the time task is running, the global configuration will be frozen, if you try to change it, an error will be raised.

3.3 Workflow for RL Experiments

In Baconian, the control flow of the experiments is delegated to an independent module `baconian.core.flow.train_test_flow:Flow` which is an abstract class. The reason to do so is to improve the flexibility and extensibility of framework. Two typical flow are implemented. One is `baconian.core.flow.train_test_flow:TrainTestFlow`, which corresponds to the pipeline of most model-free algorithms, which is sampling-training-testing pipeline. The other one is `baconian.core.flow.dyna_flow.py:DynaFlow`, which is the flow in Dyna algorithm [Sutton, 1992].

A typical train-test flow follows the diagram below:



- Train/Test condition: **(t > x) and (t – last train/test time step > n)**
- End condition: **t > N**
- **t is the current time step, called a timer.**
  - usually is the number of data the agent gets from environment.
  - can be changed to other timer if it's monotonic, e.g., total times the agent execute training.
- x represents start the train/test at least after x time steps
- n represents trigger the train/test every n time steps.
- N is the max value for the time steps.

Note: The reason why we add this module with these parameters settings is to enable the user to fully customize the experiment execution process.

For setting the end point N in the flow, see the section you can achieve by setting the global configuration in following code example. For using other status as end point, see section *Built-in Global Status/Counter* for more built-in status.

A flow module is required to pass in one step function as the timer to control the process. More detailed explanations of the time step function are given below.

For `baconian.core.flow.dyna_flow.py:DynaFlow`, comparing to `baconian.core.flow.train_test_flow:TrainTestFlow`, there are three more processes are added: `train_agent_from_model`, `train_dynamics_model`, and `test_dynamics_model`. They follows the similar ideas on how to control these processes, user can refer to its usage and example codes for more details.

3.4 Time Step Function

RL experiment often relies on a timer/counter to indicate the progress of the experiment. It can be used to schedule the parameters from modules like action noise, exploration strategy. It can also be used to timestamp a recorded log like the evaluated performance of agent so you can know the changes of the performance along with the different timestep.

Naturally, the number of samples generated from environment for training purpose is used as the time step function. This value can be retrieved from following code. Or you can built the time step function using any status value from any objects (i.e., agent, env, algorithm) as long as it is a monotonic counter.

For using built-in agent status counter as the time step function, you can see more in the following section.

3.5 Built-in Global Status/Counter

We include some recorded values as a global shared status which can be accessed during the experiments.

The all built-in recorded values are listed below:

- TOTAL_AGENT_TRAIN_SAMPLE_COUNT: the timesteps/samples used by agent for training
- TOTAL_AGENT_TRAIN_SAMPLE_FUNC_COUNT: the times of sampling function called by agent during training
- TOTAL_AGENT_TEST_SAMPLE_COUNT: the timesteps/samples used by agent for testing
- TOTAL_AGENT_UPDATE_COUNT: the times of training function called by agent
- TOTAL_ENV_STEP_TRAIN_SAMPLE_COUNT: the timesteps used by environment during training, it differs a little from TOTAL_AGENT_TRAIN_SAMPLE_COUNT
- TOTAL_ENV_STEP_TEST_SAMPLE_COUNT: the timesteps used by environment during testing, it differs a little from TOTAL_AGENT_TEST_SAMPLE_COUNT

User can access these values from anywhere they want or register new global status into by following code snippet after you start to execute the experiment:

```
from baconian.core.status import get_global_status_collect
# access directly
print(get_global_status_collect()('TOTAL_AGENT_TRAIN_SAMPLE_COUNT'))
# wrap in to a function
```

(continues on next page)

(continued from previous page)

```
train_sample_count_func=lambda: get_global_status_collect()('TOTAL_AGENT_TRAIN_SAMPLE_
↪COUNT')

# register new status
get_global_status_collect().register_info_key_status(
                                                      # object that hold the source␣
↪value
                                                     obj=object,
                                                      # which key the object used
                                                     info_key='predict_counter',
                                                     # under which status
                                                     under_status='TRAIN',
                                                     # name used to store in global␣
↪status
                                                     return_name='TOTAL_AGENT_TRAIN_
↪SAMPLE_COUNT'
                                                     )
```

Example Usages:

1. Use it to decide when to end the experiment using DEFAULT_EXPERIMENT_END_POINT in Global Config User can set by:

```
# only set one value with its limit as the end point.
# once the recorded value TOTAL_AGENT_TRAIN_SAMPLE_COUNT exceed 200, the experiment␣
↪will end.
GlobalConfig().set('DEFAULT_EXPERIMENT_END_POINT', dict(TOTAL_AGENT_TRAIN_SAMPLE_
↪COUNT=200))
```

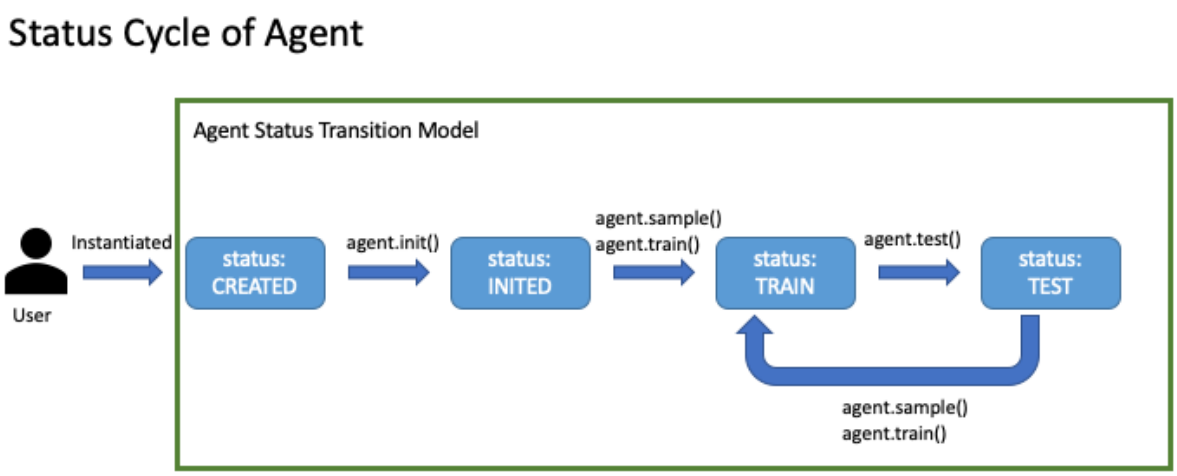
2. Access the value as the scheduler parameters' clock:

See Page *Scheduler Parameters*.

3.6 Stateful Behaviour

Status control is a must for DRL experiments. For instance, off-policy DRL methods need to switch between behavior policy and target policy during sampling and testing or decay the exploration action noise w.r.t the training progress status.

The following figure shows the status transition of agent and how it control the behaviour of agent.



- agent.init(): initialize neural network parameters, warm up the algorithm (e.g., sample a batch of data to get the mean and variance of states)
- agent.sample(), agent.train(): sample the data under the TRAIN status, and append the data into the memory. Train agent's algorithm using data from memory. Under this status, agent will enable the exploration strategy, action noise, etc.
- agent.test(): test the agent, the exploration strategy/action noise, etc. will be disabled.

Every class that inherited from `baconian.core.core:Basic` will have two class attributes: `STATUS_LIST` which contains all status of this class or module.

You can call `set_status` method to change the status of one instance. You can call `get_status` method to get the current status of an instance, which is a dict type. The return value not only contains the status (i.e., TRAIN, TEST) but also other extra information that is specially added in the code. Such as, you can register a counter of a function by doing so:

```
        else:
            if self.noise_adder and not self.is_testing:
                res = self.env_spec.action_space.clip(self.noise_adder(self.algo.
↪predict(**kwargs)))
            else:
                res = self.algo.predict(**kwargs)
        self.recorder.append_to_obj_log(obj=self, attr_name='action', status_
↪info=self.get_status(), value=res)
        return res

    @register_counter_info_to_status_decorator(increment=1, info_key='sample_counter',
↪ under_status=('TRAIN', 'TEST'),
                                               ignore_wrong_status=True)
    def sample(self, env, sample_count: int, in_which_status: str = 'TRAIN', store_
↪flag=False,
               sample_type: str = 'transition') -> (
            TransitionData, TrajectoryData):
        """
        sample a certain number of data from environment

        :param env: environment to sample
        :param sample_count: int, sample count
        :param in_which_status: string, environment status
        :param store_flag: to store environment samples or not, default False
```

The counter for calling function `predict` is added as one attribute of status, which will be returned with the key

`predict_counter`

For detailed usage of these methods, you can find it in API reference.

CHAPTER 4

Examples

We offer some examples on how to better use Baconian in a more practical and efficient way.

4.1 DQN with Acrobot-v1

```
from baconian.algo.dqn import DQN
from baconian.core.core import EnvSpec
from baconian.envs.gym_env import make
from baconian.algo.value_func.mlp_q_value import MLPQValueFunction
from baconian.core.agent import Agent
from baconian.algo.misc import EpsilonGreedy
from baconian.core.experiment import Experiment
from baconian.core.flow.train_test_flow import create_train_test_flow
from baconian.config.global_config import GlobalConfig
from baconian.common.schedules import LinearScheduler
from baconian.core.status import get_global_status_collect


def task_fn():
    env = make('Acrobot-v1')
    name = 'demo_exp'
    env_spec = EnvSpec(obs_space=env.observation_space,
                       action_space=env.action_space)
    mlp_q = MLPQValueFunction(env_spec=env_spec,
                              name_scope=name + '_mlp_q',
                              name=name + '_mlp_q',
                              mlp_config=[
                                  {
                                      "ACT": "TANH",
                                      "B_INIT_VALUE": 0.0,
                                      "NAME": "1",
                                      "N_UNITS": 64,
                                      "TYPE": "DENSE",
```

(continues on next page)

(continued from previous page)

```
                                    "W_NORMAL_STDDEV": 0.03
                                },
                                {
                                    "ACT": "TANH",
                                    "B_INIT_VALUE": 0.0,
                                    "NAME": "2",
                                    "N_UNITS": 64,
                                    "TYPE": "DENSE",
                                    "W_NORMAL_STDDEV": 0.03
                                },
                                {
                                    "ACT": "RELU",
                                    "B_INIT_VALUE": 0.0,
                                    "NAME": "3",
                                    "N_UNITS": 256,
                                    "TYPE": "DENSE",
                                    "W_NORMAL_STDDEV": 0.03
                                },
                                {
                                    "ACT": "LINEAR",
                                    "B_INIT_VALUE": 0.0,
                                    "NAME": "OUPTUT",
                                    "N_UNITS": 1,
                                    "TYPE": "DENSE",
                                    "W_NORMAL_STDDEV": 0.03
                                }
                            ])
    dqn = DQN(env_spec=env_spec,
              config_or_config_dict=dict(REPLAY_BUFFER_SIZE=50000,
                                         GAMMA=0.99,
                                         BATCH_SIZE=32,
                                         LEARNING_RATE=0.001,
                                         TRAIN_ITERATION=1,
                                         DECAY=0),
              name=name + '_dqn',
              value_func=mlp_q)
    agent = Agent(env=env, env_spec=env_spec,
                  algo=dqn,
                  name=name + '_agent',
                  exploration_strategy=EpsilonGreedy(action_space=env_spec.action_
↪space,
                                                     prob_scheduler=LinearScheduler(
                                                         t_fn=lambda: get_global_
↪status_collect()(
                                                             'TOTAL_AGENT_TRAIN_
↪SAMPLE_COUNT'),
                                                         schedule_timesteps=int(0.1 *␣
↪100000),
                                                         initial_p=1.0,
                                                         final_p=0.02),
                                                     init_random_prob=0.1),
                  noise_adder=None)

    flow = create_train_test_flow(
        test_every_sample_count=100,
        train_every_sample_count=1,
        start_test_after_sample_count=0,
```

(continues on next page)

(continued from previous page)

```
        start_train_after_sample_count=1000,
        sample_func_and_args=(agent.sample, (), dict(sample_count=1,
                                                     env=agent.env,
                                                     store_flag=True)),
        train_func_and_args=(agent.train, (), dict()),
        test_func_and_args=(agent.test, (), dict(sample_count=3)),
    )
    experiment = Experiment(
        tuner=None,
        env=env,
        agent=agent,
        flow=flow,
        name=name
    )
    experiment.run()


from baconian.core.experiment_runner import *

GlobalConfig().set('DEFAULT_LOG_PATH', './log_path')
single_exp_runner(task_fn, del_if_log_path_existed=True)
```

4.2 DDPG with Pendulum-v0

```
"""
A simple example to show how to build up an experiment with ddpg training and testing␣
↪on Pendulum-v0
"""
from baconian.core.core import EnvSpec
from baconian.envs.gym_env import make
from baconian.algo.value_func.mlp_q_value import MLPQValueFunction
from baconian.algo.ddpg import DDPG
from baconian.algo.policy import DeterministicMLPPolicy
from baconian.core.agent import Agent
from baconian.algo.misc import EpsilonGreedy
from baconian.core.experiment import Experiment
from baconian.core.flow.train_test_flow import create_train_test_flow
from baconian.config.global_config import GlobalConfig
from baconian.core.status import get_global_status_collect
from baconian.common.schedules import PeriodicalEventSchedule
import baconian.common.log_data_loader as loader
from pathlib import Path


def task_fn():
    env = make('Pendulum-v0')
    name = 'demo_exp'
    env_spec = EnvSpec(obs_space=env.observation_space,
                       action_space=env.action_space)

    mlp_q = MLPQValueFunction(env_spec=env_spec,
                              name_scope=name + '_mlp_q',
                              name=name + '_mlp_q',
                              mlp_config=[
```

(continues on next page)

(continued from previous page)

```
                                    {
                                        "ACT": "RELU",
                                        "B_INIT_VALUE": 0.0,
                                        "NAME": "1",
                                        "N_UNITS": 16,
                                        "TYPE": "DENSE",
                                        "W_NORMAL_STDDEV": 0.03
                                    },
                                    {
                                        "ACT": "LINEAR",
                                        "B_INIT_VALUE": 0.0,
                                        "NAME": "OUPTUT",
                                        "N_UNITS": 1,
                                        "TYPE": "DENSE",
                                        "W_NORMAL_STDDEV": 0.03
                                    }
                                ])
    policy = DeterministicMLPPolicy(env_spec=env_spec,
                                    name_scope=name + '_mlp_policy',
                                    name=name + '_mlp_policy',
                                    mlp_config=[
                                        {
                                            "ACT": "RELU",
                                            "B_INIT_VALUE": 0.0,
                                            "NAME": "1",
                                            "N_UNITS": 16,
                                            "TYPE": "DENSE",
                                            "W_NORMAL_STDDEV": 0.03
                                        },
                                        {
                                            "ACT": "LINEAR",
                                            "B_INIT_VALUE": 0.0,
                                            "NAME": "OUPTUT",
                                            "N_UNITS": env_spec.flat_action_dim,
                                            "TYPE": "DENSE",
                                            "W_NORMAL_STDDEV": 0.03
                                        }
                                    ],
                                    reuse=False)

    ddpg = DDPG(
        env_spec=env_spec,
        config_or_config_dict={
            "REPLAY_BUFFER_SIZE": 10000,
            "GAMMA": 0.999,
            "CRITIC_LEARNING_RATE": 0.001,
            "ACTOR_LEARNING_RATE": 0.001,
            "DECAY": 0.5,
            "BATCH_SIZE": 50,
            "TRAIN_ITERATION": 1,
            "critic_clip_norm": 0.1,
            "actor_clip_norm": 0.1,
        },
        value_func=mlp_q,
        policy=policy,
        name=name + '_ddpg',
        replay_buffer=None
```

(continues on next page)

(continued from previous page)

```
    )
    agent = Agent(env=env, env_spec=env_spec,
                  algo=ddpg,
                  algo_saving_scheduler=PeriodicalEventSchedule(
                      t_fn=lambda: get_global_status_collect()('TOTAL_AGENT_TRAIN_
↪SAMPLE_COUNT'),
                      trigger_every_step=20,
                      after_t=10),
                  name=name + '_agent',
                  exploration_strategy=EpsilonGreedy(action_space=env_spec.action_
↪space,
                                                     init_random_prob=0.5))

    flow = create_train_test_flow(
        test_every_sample_count=10,
        train_every_sample_count=10,
        start_test_after_sample_count=5,
        start_train_after_sample_count=5,
        train_func_and_args=(agent.train, (), dict()),
        test_func_and_args=(agent.test, (), dict(sample_count=1)),
        sample_func_and_args=(agent.sample, (), dict(sample_count=100,
                                                     env=agent.env,
                                                     store_flag=True))
    )

    experiment = Experiment(
        tuner=None,
        env=env,
        agent=agent,
        flow=flow,
        name=name
    )
    experiment.run()


from baconian.core.experiment_runner import *

GlobalConfig().set('DEFAULT_LOG_PATH', './log_path')
single_exp_runner(task_fn, del_if_log_path_existed=True)
```

4.3 PPO with Pendulum-v0

```
# Date: 3/30/19
# Author: Luke
# Project: baconian-internal

"""
A simple example to show how to build up an experiment with ppo training and testing␣
↪on Pendulum-v0
"""
from baconian.core.core import EnvSpec
from baconian.envs.gym_env import make
from baconian.algo.value_func import MLPVValueFunc
from baconian.algo.ppo import PPO
```

(continues on next page)

(continued from previous page)

```
from baconian.algo.policy.normal_distribution_mlp import NormalDistributionMLPPolicy
from baconian.core.agent import Agent
from baconian.algo.misc import EpsilonGreedy
from baconian.core.experiment import Experiment
from baconian.core.flow.train_test_flow import create_train_test_flow
from baconian.config.global_config import GlobalConfig
from baconian.core.status import get_global_status_collect
from baconian.common.schedules import PeriodicalEventSchedule


def task_fn():
    env = make('Pendulum-v0')
    name = 'demo_exp_'
    env_spec = EnvSpec(obs_space=env.observation_space,
                       action_space=env.action_space)

    mlp_v = MLPVValueFunc(env_spec=env_spec,
                          name_scope=name + 'mlp_v',
                          name=name + 'mlp_v',
                          mlp_config=[
                              {
                                  "ACT": "RELU",
                                  "B_INIT_VALUE": 0.0,
                                  "NAME": "1",
                                  "N_UNITS": 16,
                                  "L1_NORM": 0.01,
                                  "L2_NORM": 0.01,
                                  "TYPE": "DENSE",
                                  "W_NORMAL_STDDEV": 0.03
                              },
                              {
                                  "ACT": "LINEAR",
                                  "B_INIT_VALUE": 0.0,
                                  "NAME": "OUPTUT",
                                  "N_UNITS": 1,
                                  "TYPE": "DENSE",
                                  "W_NORMAL_STDDEV": 0.03
                              }
                          ])

    policy = NormalDistributionMLPPolicy(env_spec=env_spec,
                                         name_scope=name + 'mlp_policy',
                                         name=name + 'mlp_policy',
                                         mlp_config=[
                                             {
                                                 "ACT": "RELU",
                                                 "B_INIT_VALUE": 0.0,
                                                 "NAME": "1",
                                                 "L1_NORM": 0.01,
                                                 "L2_NORM": 0.01,
                                                 "N_UNITS": 16,
                                                 "TYPE": "DENSE",
                                                 "W_NORMAL_STDDEV": 0.03
                                             },
                                             {
                                                 "ACT": "LINEAR",
                                                 "B_INIT_VALUE": 0.0,
```

(continues on next page)

(continued from previous page)

```
                                                "NAME": "OUPTUT",
                                                "N_UNITS": env_spec.flat_action_dim,
                                                "TYPE": "DENSE",
                                                "W_NORMAL_STDDEV": 0.03
                                            }
                                        ],
                                        reuse=False)

    ppo = PPO(
        env_spec=env_spec,
        config_or_config_dict={
            "gamma": 0.995,
            "lam": 0.98,
            "policy_train_iter": 10,
            "value_func_train_iter": 10,
            "clipping_range": None,
            "beta": 1.0,
            "eta": 50,
            "value_func_memory_size": 10,
            "log_var_init": -1.0,
            "kl_target": 0.003,
            "policy_lr": 0.01,
            "value_func_lr": 0.01,
            "value_func_train_batch_size": 10,
            "lr_multiplier": 1.0
        },
        value_func=mlp_v,
        stochastic_policy=policy,
        name=name + 'ppo'
    )
    agent = Agent(env=env, env_spec=env_spec,
                  algo=ppo,
                  algo_saving_scheduler=PeriodicalEventSchedule(
                      t_fn=lambda: get_global_status_collect()('TOTAL_AGENT_TRAIN_
↪SAMPLE_COUNT'),
                      trigger_every_step=20,
                      after_t=10),
                  name=name + 'agent',
                  exploration_strategy=EpsilonGreedy(action_space=env_spec.action_
↪space,
                                                     init_random_prob=0.5))
    flow = create_train_test_flow(
        test_every_sample_count=10,
        train_every_sample_count=10,
        start_test_after_sample_count=5,
        start_train_after_sample_count=5,
        train_func_and_args=(agent.train, (), dict()),
        test_func_and_args=(agent.test, (), dict(sample_count=10)),
        sample_func_and_args=(agent.sample, (), dict(sample_count=100,
                                                     env=agent.env,
                                                     sample_type='trajectory',
                                                     store_flag=True))
    )

    experiment = Experiment(
        tuner=None,
        env=env,
```

(continues on next page)

(continued from previous page)

```
        agent=agent,
        flow=flow,
        name=name
    )
    experiment.run()


from baconian.core.experiment_runner import single_exp_runner

GlobalConfig().set('DEFAULT_LOG_PATH', './log_path')
single_exp_runner(task_fn, del_if_log_path_existed=True)
```

4.4 MPC with Pendulum-v0

```
"""
A simple example to show how to build up an experiment with ddpg training and testing␣
↪on MountainCarContinuous-v0
"""
from baconian.core.core import EnvSpec
from baconian.envs.gym_env import make
from baconian.core.agent import Agent
from baconian.algo.misc import EpsilonGreedy
from baconian.core.experiment import Experiment
from baconian.core.flow.train_test_flow import create_train_test_flow
from baconian.algo.mpc import ModelPredictiveControl
from baconian.algo.dynamics.terminal_func.terminal_func import RandomTerminalFunc
from baconian.algo.dynamics.reward_func.reward_func import RandomRewardFunc
from baconian.algo.policy import UniformRandomPolicy
from baconian.algo.dynamics.mlp_dynamics_model import ContinuousMLPGlobalDynamicsModel
from baconian.config.global_config import GlobalConfig


def task_fn():
    env = make('Pendulum-v0')
    name = 'demo_exp'
    env_spec = EnvSpec(obs_space=env.observation_space,
                       action_space=env.action_space)

    mlp_dyna = ContinuousMLPGlobalDynamicsModel(
        env_spec=env_spec,
        name_scope=name + '_mlp_dyna',
        name=name + '_mlp_dyna',
        learning_rate=0.01,
        mlp_config=[
            {
                "ACT": "RELU",
                "B_INIT_VALUE": 0.0,
                "NAME": "1",
                "L1_NORM": 0.0,
                "L2_NORM": 0.0,
                "N_UNITS": 16,
                "TYPE": "DENSE",
                "W_NORMAL_STDDEV": 0.03
            },
```

(continues on next page)

(continued from previous page)

```
            {
                "ACT": "LINEAR",
                "B_INIT_VALUE": 0.0,
                "NAME": "OUPTUT",
                "L1_NORM": 0.0,
                "L2_NORM": 0.0,
                "N_UNITS": env_spec.flat_obs_dim,
                "TYPE": "DENSE",
                "W_NORMAL_STDDEV": 0.03
            }
        ])
    algo = ModelPredictiveControl(
        dynamics_model=mlp_dyna,
        env_spec=env_spec,
        config_or_config_dict=dict(
            SAMPLED_HORIZON=2,
            SAMPLED_PATH_NUM=5,
            dynamics_model_train_iter=10
        ),
        name=name + '_mpc',

        policy=UniformRandomPolicy(env_spec=env_spec, name='uni_policy')
    )
    algo.set_terminal_reward_function_for_dynamics_env(reward_
↪func=RandomRewardFunc(name='reward_func'),
                                                       terminal_
↪func=RandomTerminalFunc(name='random_terminal'), )
    agent = Agent(env=env, env_spec=env_spec,
                  algo=algo,
                  name=name + '_agent',
                  exploration_strategy=EpsilonGreedy(action_space=env_spec.action_
↪space,
                                                     init_random_prob=0.5))
    flow = create_train_test_flow(
        test_every_sample_count=10,
        train_every_sample_count=10,
        start_test_after_sample_count=5,
        start_train_after_sample_count=5,
        train_func_and_args=(agent.train, (), dict()),
        test_func_and_args=(agent.test, (), dict(sample_count=10)),
        sample_func_and_args=(agent.sample, (), dict(sample_count=100,
                                                     env=agent.env,
                                                     store_flag=True))
    )
    experiment = Experiment(
        tuner=None,
        env=env,
        agent=agent,
        flow=flow,
        name=name
    )
    experiment.run()


from baconian.core.experiment_runner import single_exp_runner

GlobalConfig().set('DEFAULT_LOG_PATH', './log_path')
```

(continues on next page)

(continued from previous page)

```
single_exp_runner(task_fn, del_if_log_path_existed=True)
```

4.5 Dyna with Pendulum-v0

```
"""
A simple example to show how to build up an experiment with Dyna training and testing␣
↪on Pendulum-v0
"""
from baconian.core.core import EnvSpec
from baconian.envs.gym_env import make
from baconian.algo.value_func.mlp_q_value import MLPQValueFunction
from baconian.algo.ddpg import DDPG
from baconian.algo.policy import DeterministicMLPPolicy
from baconian.core.agent import Agent
from baconian.algo.misc import EpsilonGreedy
from baconian.core.experiment import Experiment
from baconian.config.global_config import GlobalConfig
from baconian.core.status import get_global_status_collect
from baconian.common.schedules import PeriodicalEventSchedule
from baconian.algo.dynamics.mlp_dynamics_model import ContinuousMLPGlobalDynamicsModel
from baconian.algo.dyna import Dyna
from baconian.algo.dynamics.reward_func.reward_func import RandomRewardFunc
from baconian.algo.dynamics.terminal_func.terminal_func import␣
↪FixedEpisodeLengthTerminalFunc
from baconian.core.flow.dyna_flow import create_dyna_flow
from baconian.common.data_pre_processing import RunningStandardScaler


def task_fn():
    # create the gym environment by make function
    env = make('Pendulum-v0')
    # give your experiment a name which is used to generate the log path etc.
    name = 'demo_exp'
    # construct the environment specification
    env_spec = EnvSpec(obs_space=env.observation_space,
                       action_space=env.action_space)
    # construct the neural network to approximate q function of DDPG
    mlp_q = MLPQValueFunction(env_spec=env_spec,
                              name_scope=name + '_mlp_q',
                              name=name + '_mlp_q',
                              mlp_config=[
                                  {
                                      "ACT": "RELU",
                                      "B_INIT_VALUE": 0.0,
                                      "NAME": "1",
                                      "N_UNITS": 16,
                                      "TYPE": "DENSE",
                                      "W_NORMAL_STDDEV": 0.03
                                  },
                                  {
                                      "ACT": "LINEAR",
                                      "B_INIT_VALUE": 0.0,
                                      "NAME": "OUPTUT",
                                      "N_UNITS": 1,
```

(continues on next page)

(continued from previous page)

```
                                    "TYPE": "DENSE",
                                    "W_NORMAL_STDDEV": 0.03
                                }
                            ])
    # construct the neural network to approximate policy for DDPG
    policy = DeterministicMLPPolicy(env_spec=env_spec,
                                    name_scope=name + '_mlp_policy',
                                    name=name + '_mlp_policy',
                                    mlp_config=[
                                        {
                                            "ACT": "RELU",
                                            "B_INIT_VALUE": 0.0,
                                            "NAME": "1",
                                            "N_UNITS": 16,
                                            "TYPE": "DENSE",
                                            "W_NORMAL_STDDEV": 0.03
                                        },
                                        {
                                            "ACT": "LINEAR",
                                            "B_INIT_VALUE": 0.0,
                                            "NAME": "OUPTUT",
                                            "N_UNITS": env_spec.flat_action_dim,
                                            "TYPE": "DENSE",
                                            "W_NORMAL_STDDEV": 0.03
                                        }
                                    ],
                                    reuse=False)
    # construct the DDPG algorithms
    ddpg = DDPG(
        env_spec=env_spec,
        config_or_config_dict={
            "REPLAY_BUFFER_SIZE": 10000,
            "GAMMA": 0.999,
            "CRITIC_LEARNING_RATE": 0.001,
            "ACTOR_LEARNING_RATE": 0.001,
            "DECAY": 0.5,
            "BATCH_SIZE": 50,
            "TRAIN_ITERATION": 1,
            "critic_clip_norm": 0.1,
            "actor_clip_norm": 0.1,
        },
        value_func=mlp_q,
        policy=policy,
        name=name + '_ddpg',
        replay_buffer=None
    )
    # construct a neural network based global dynamics model to approximate the state␣
↪transition of environment
    mlp_dyna = ContinuousMLPGlobalDynamicsModel(
        env_spec=env_spec,
        name_scope=name + '_mlp_dyna',
        name=name + '_mlp_dyna',
        learning_rate=0.01,
        state_input_scaler=RunningStandardScaler(dims=env_spec.flat_obs_dim),
        action_input_scaler=RunningStandardScaler(dims=env_spec.flat_action_dim),
        output_delta_state_scaler=RunningStandardScaler(dims=env_spec.flat_obs_dim),
        mlp_config=[
```

(continues on next page)

(continued from previous page)

```
            {
                "ACT": "RELU",
                "B_INIT_VALUE": 0.0,
                "NAME": "1",
                "L1_NORM": 0.0,
                "L2_NORM": 0.0,
                "N_UNITS": 16,
                "TYPE": "DENSE",
                "W_NORMAL_STDDEV": 0.03
            },
            {
                "ACT": "LINEAR",
                "B_INIT_VALUE": 0.0,
                "NAME": "OUPTUT",
                "L1_NORM": 0.0,
                "L2_NORM": 0.0,
                "N_UNITS": env_spec.flat_obs_dim,
                "TYPE": "DENSE",
                "W_NORMAL_STDDEV": 0.03
            }
        ])
    # finally, construct the Dyna algorithms with a model free algorithm DDGP, and a␣
↪NN model.
    algo = Dyna(env_spec=env_spec,
                name=name + '_dyna_algo',
                model_free_algo=ddpg,
                dynamics_model=mlp_dyna,
                config_or_config_dict=dict(
                    dynamics_model_train_iter=10,
                    model_free_algo_train_iter=10
                ))
    # To make the NN based dynamics model a proper environment so be a sampling␣
↪source for DDPG, reward function and
    # terminal function need to be set.

    # For examples only, we use random reward function and terminal function with␣
↪fixed episode length.
    algo.set_terminal_reward_function_for_dynamics_env(
        terminal_func=FixedEpisodeLengthTerminalFunc(max_step_length=env.unwrapped._
↪max_episode_steps,
                                                     step_count_fn=algo.dynamics_env.
↪total_step_count_fn),
        reward_func=RandomRewardFunc())
    # construct agent with additional exploration strategy if needed.
    agent = Agent(env=env, env_spec=env_spec,
                  algo=algo,
                  algo_saving_scheduler=PeriodicalEventSchedule(
                      t_fn=lambda: get_global_status_collect()('TOTAL_AGENT_TRAIN_
↪SAMPLE_COUNT'),
                      trigger_every_step=20,
                      after_t=10),
                  name=name + '_agent',
                  exploration_strategy=EpsilonGreedy(action_space=env_spec.action_
↪space,
                                                     init_random_prob=0.5))
    # construct the training flow, called Dyna flow. It defines how the training␣
↪proceed, and the terminal condition
```

(continues on next page)

(continued from previous page)

```
    flow = create_dyna_flow(
        train_algo_func=(agent.train, (), dict(state='state_agent_training')),
        train_algo_from_synthesized_data_func=(agent.train, (), dict(state='state_
↪agent_training')),
        train_dynamics_func=(agent.train, (), dict(state='state_dynamics_training')),
        test_algo_func=(agent.test, (), dict(sample_count=1)),
        test_dynamics_func=(agent.algo.test_dynamics, (), dict(sample_count=10,␣
↪env=env)),
        sample_from_real_env_func=(agent.sample, (), dict(sample_count=10,
                                                          env=agent.env,
                                                          store_flag=True)),
        sample_from_dynamics_env_func=(agent.sample, (), dict(sample_count=10,
                                                              env=agent.algo.dynamics_
↪env,
                                                              store_flag=True)),
        train_algo_every_real_sample_count_by_data_from_real_env=40,
        train_algo_every_real_sample_count_by_data_from_dynamics_env=40,
        test_algo_every_real_sample_count=40,
        test_dynamics_every_real_sample_count=40,
        train_dynamics_ever_real_sample_count=20,
        start_train_algo_after_sample_count=1,
        start_train_dynamics_after_sample_count=1,
        start_test_algo_after_sample_count=1,
        start_test_dynamics_after_sample_count=1,
        warm_up_dynamics_samples=1
    )
    # construct the experiment
    experiment = Experiment(
        tuner=None,
        env=env,
        agent=agent,
        flow=flow,
        name=name + '_exp'
    )
    # run!
    experiment.run()


from baconian.core.experiment_runner import *

# set some global configuration here

# set DEFAULT_EXPERIMENT_END_POINT to indicate when to stop the experiment.
# one usually used is the TOTAL_AGENT_TRAIN_SAMPLE_COUNT, i.e., how many samples/
↪timesteps are used for training
GlobalConfig().set('DEFAULT_EXPERIMENT_END_POINT', dict(TOTAL_AGENT_TRAIN_SAMPLE_
↪COUNT=200))

# set the logging path to write log and save model checkpoints.
GlobalConfig().set('DEFAULT_LOG_PATH', './log_path')

# feed the task into a exp runner.
single_exp_runner(task_fn, del_if_log_path_existed=True)
```

4.6 Gaussian Process Dynamics

```
"""
This gives a simple example on how to use Gaussian Process (GP) to approximate the␣
↪Gym environment Pendulum-v0
We use gpflow package to build the Gaussian Process.
"""
from baconian.core.core import EnvSpec
from baconian.envs.gym_env import make
import numpy as np
from baconian.common.sampler.sample_data import TransitionData
from baconian.algo.policy import UniformRandomPolicy
from baconian.algo.dynamics.gaussian_process_dynamiocs_model import␣
↪GaussianProcessDyanmicsModel
from baconian.algo.dynamics.dynamics_model import DynamicsEnvWrapper
from baconian.algo.dynamics.terminal_func.terminal_func import RandomTerminalFunc
from baconian.algo.dynamics.reward_func.reward_func import RandomRewardFunc

env = make('Pendulum-v0')
name = 'demo_exp'
env_spec = EnvSpec(obs_space=env.observation_space,
                   action_space=env.action_space)
data = TransitionData(env_spec=env_spec)
policy = UniformRandomPolicy(env_spec=env_spec)
# Do some initial sampling here to train GP model
st = env.reset()
for i in range(100):
    ac = policy.forward(st)
    new_st, re, _, _ = env.step(ac)
    data.append(state=st, new_state=new_st, action=ac, reward=re, done=False)
    st = new_st

gp = GaussianProcessDyanmicsModel(env_spec=env_spec, batch_data=data)
gp.init()
gp.train()

dyna_env = DynamicsEnvWrapper(dynamics=gp)
# Since we only care about the prediction here, so we pass the terminal function and␣
↪reward function setting with
# random one
dyna_env.set_terminal_reward_func(terminal_func=RandomTerminalFunc(),
                                  reward_func=RandomRewardFunc())

st = env.reset()
real_state_list = []
dynamics_state_list = []
test_sample_count = 100
for i in range(test_sample_count):
    ac = env_spec.action_space.sample()
    gp.reset_state(state=st)
    new_state_dynamics, _, _, _ = dyna_env.step(action=ac, allow_clip=True)
    new_state_real, _, done, _ = env.step(action=ac)
    real_state_list.append(new_state_real)
    dynamics_state_list.append(new_state_dynamics)
    st = new_state_real
    if done is True:
        env.reset()
```

(continues on next page)

(continued from previous page)

```
l1_loss = np.linalg.norm(np.array(real_state_list) - np.array(dynamics_state_list),␣
↪ord=1)
l2_loss = np.linalg.norm(np.array(real_state_list) - np.array(dynamics_state_list),␣
↪ord=2)
print('l1 loss is {}, l2 loss is {}'.format(l1_loss, l2_loss))
```

4.7 Early Stopping Flow (DDPG with Pendulum-v0)

```
"""
This script show the example for adding an early stopping feature so when the agent␣
↪can't increase its received average
reward for evaluation, the experiment will end early.

To do so in a extensible and modular way. We can implement a new flow called␣
↪EarlyStoppingFlow that implement a special
ending condition detections by accessing the agent's evaluation reward (with built-in␣
↪modular to access). Such mechanism
can be re-used by all algorithms, which avoid the redundant coding for users.
"""
from baconian.config.dict_config import DictConfig
from baconian.core.flow.train_test_flow import TrainTestFlow

from baconian.core.core import EnvSpec
from baconian.envs.gym_env import make
from baconian.algo.value_func.mlp_q_value import MLPQValueFunction
from baconian.algo.ddpg import DDPG
from baconian.algo.policy import DeterministicMLPPolicy
from baconian.core.agent import Agent
from baconian.algo.misc import EpsilonGreedy
from baconian.core.experiment import Experiment
from baconian.core.status import get_global_status_collect
from baconian.common.schedules import PeriodicalEventSchedule


class EarlyStoppingFlow(TrainTestFlow):
    required_key_dict = {
        **TrainTestFlow.required_key_dict,
        'USE_LAST_K_EVALUATION_REWARD': 10
    }

    def __init__(self, train_sample_count_func, config_or_config_dict: (DictConfig,␣
↪dict), func_dict: dict, agent):
        super().__init__(train_sample_count_func, config_or_config_dict, func_dict)
        self.agent = agent

    def _is_ended(self):
        test_reward = sorted(self.agent.recorder.get_log(attr_name='sum_reward',␣
↪filter_by_status=dict(status='TEST')),
                             key=lambda x: x['sample_counter'])
        if len(test_reward) >= self.parameters('USE_LAST_K_EVALUATION_REWARD') * 2:
            last_reward = test_reward[-self.parameters('USE_LAST_K_EVALUATION_REWARD
↪'):]
            pre_reward = test_reward[-self.parameters('USE_LAST_K_EVALUATION_REWARD')␣
↪* 2: -self.parameters(
```

(continues on next page)

(continued from previous page)

```
                'USE_LAST_K_EVALUATION_REWARD')]
            last_reward = np.mean([r['value'] for r in last_reward])
            pre_reward = np.mean([r['value'] for r in pre_reward])
            if last_reward < pre_reward:
                ConsoleLogger().print('info', 'training ended because last {} step␣
↪reward: {} < previous {} step reward {}'.format(self.parameters('USE_LAST_K_
↪EVALUATION_REWARD'), last_reward, self.parameters('USE_LAST_K_EVALUATION_REWARD'),␣
↪pre_reward))
                return True
        return super()._is_ended()


def create_early_stopping_flow(test_every_sample_count, train_every_sample_count,␣
↪start_train_after_sample_count,
                               start_test_after_sample_count, train_func_and_args,␣
↪test_func_and_args,
                               sample_func_and_args,
                               agent,
                               use_last_k_evaluation_reward,
                               train_samples_counter_func=None):
    config_dict = dict(
        TEST_EVERY_SAMPLE_COUNT=test_every_sample_count,
        TRAIN_EVERY_SAMPLE_COUNT=train_every_sample_count,
        START_TRAIN_AFTER_SAMPLE_COUNT=start_train_after_sample_count,
        START_TEST_AFTER_SAMPLE_COUNT=start_test_after_sample_count,
        USE_LAST_K_EVALUATION_REWARD=use_last_k_evaluation_reward
    )

    def return_func_dict(s_dict):
        return dict(func=s_dict[0],
                    args=s_dict[1],
                    kwargs=s_dict[2])

    func_dict = dict(
        train=return_func_dict(train_func_and_args),
        test=return_func_dict(test_func_and_args),
        sample=return_func_dict(sample_func_and_args),
    )
    if train_samples_counter_func is None:
        def default_train_samples_counter_func():
            return get_global_status_collect()('TOTAL_AGENT_TRAIN_SAMPLE_COUNT')

        train_samples_counter_func = default_train_samples_counter_func

    return EarlyStoppingFlow(config_or_config_dict=config_dict,
                             train_sample_count_func=train_samples_counter_func,
                             agent=agent,
                             func_dict=func_dict)


def task_fn():
    env = make('Pendulum-v0')
    name = 'demo_exp'
    env_spec = EnvSpec(obs_space=env.observation_space,
                       action_space=env.action_space)

    mlp_q = MLPQValueFunction(env_spec=env_spec,
```

(continues on next page)

(continued from previous page)

```
                                name_scope=name + '_mlp_q',
                                name=name + '_mlp_q',
                                mlp_config=[
                                    {
                                        "ACT": "RELU",
                                        "B_INIT_VALUE": 0.0,
                                        "NAME": "1",
                                        "N_UNITS": 16,
                                        "TYPE": "DENSE",
                                        "W_NORMAL_STDDEV": 0.03
                                    },
                                    {
                                        "ACT": "LINEAR",
                                        "B_INIT_VALUE": 0.0,
                                        "NAME": "OUPTUT",
                                        "N_UNITS": 1,
                                        "TYPE": "DENSE",
                                        "W_NORMAL_STDDEV": 0.03
                                    }
                                ])
    policy = DeterministicMLPPolicy(env_spec=env_spec,
                                     name_scope=name + '_mlp_policy',
                                     name=name + '_mlp_policy',
                                     mlp_config=[
                                         {
                                             "ACT": "RELU",
                                             "B_INIT_VALUE": 0.0,
                                             "NAME": "1",
                                             "N_UNITS": 16,
                                             "TYPE": "DENSE",
                                             "W_NORMAL_STDDEV": 0.03
                                         },
                                         {
                                             "ACT": "LINEAR",
                                             "B_INIT_VALUE": 0.0,
                                             "NAME": "OUPTUT",
                                             "N_UNITS": env_spec.flat_action_dim,
                                             "TYPE": "DENSE",
                                             "W_NORMAL_STDDEV": 0.03
                                         }
                                     ],
                                     reuse=False)

    ddpg = DDPG(
        env_spec=env_spec,
        config_or_config_dict={
            "REPLAY_BUFFER_SIZE": 10000,
            "GAMMA": 0.999,
            "CRITIC_LEARNING_RATE": 0.001,
            "ACTOR_LEARNING_RATE": 0.001,
            "DECAY": 0.5,
            "BATCH_SIZE": 50,
            "TRAIN_ITERATION": 1,
            "critic_clip_norm": 0.1,
            "actor_clip_norm": 0.1,
        },
        value_func=mlp_q,
```

(continues on next page)

(continued from previous page)

```
        policy=policy,
        name=name + '_ddpg',
        replay_buffer=None
    )
    agent = Agent(env=env, env_spec=env_spec,
                  algo=ddpg,
                  algo_saving_scheduler=PeriodicalEventSchedule(
                      t_fn=lambda: get_global_status_collect()('TOTAL_AGENT_TRAIN_
↪SAMPLE_COUNT'),
                      trigger_every_step=20,
                      after_t=10),
                  name=name + '_agent',
                  exploration_strategy=EpsilonGreedy(action_space=env_spec.action_
↪space,
                                                     init_random_prob=0.5))

    flow = create_early_stopping_flow(
        agent=agent,
        use_last_k_evaluation_reward=5,
        test_every_sample_count=10,
        train_every_sample_count=10,
        start_test_after_sample_count=5,
        start_train_after_sample_count=5,
        train_func_and_args=(agent.train, (), dict()),
        test_func_and_args=(agent.test, (), dict(sample_count=1)),
        sample_func_and_args=(agent.sample, (), dict(sample_count=100,
                                                     env=agent.env))
    )

    experiment = Experiment(
        tuner=None,
        env=env,
        agent=agent,
        flow=flow,
        name=name
    )
    experiment.run()


from baconian.core.experiment_runner import *

GlobalConfig().set('DEFAULT_LOG_PATH', './log_path')
GlobalConfig().set('DEFAULT_EXPERIMENT_END_POINT',
                   dict(TOTAL_AGENT_TRAIN_SAMPLE_COUNT=2000,
                        TOTAL_AGENT_TEST_SAMPLE_COUNT=None,
                        TOTAL_AGENT_UPDATE_COUNT=None))
single_exp_runner(task_fn, del_if_log_path_existed=True)
```

4.8 Environment Wrapper (MountainCarContinuous-v0, RewardWrapper)

4.9 Use scheduler module in experiments

```
"""
In this example, we demonstrate how to utilize the scheduler module to dynamically␣
↪setting the
learning rate of your algorithm, or epsilon-greedy probability
"""

from baconian.algo.dqn import DQN
from baconian.core.core import EnvSpec
from baconian.envs.gym_env import make
from baconian.algo.value_func.mlp_q_value import MLPQValueFunction
from baconian.core.agent import Agent
from baconian.algo.misc import EpsilonGreedy
from baconian.core.experiment import Experiment
from baconian.core.flow.train_test_flow import create_train_test_flow
from baconian.config.global_config import GlobalConfig
from baconian.common.schedules import LinearScheduler, PiecewiseScheduler,␣
↪PeriodicalEventSchedule
from baconian.core.status import get_global_status_collect


def task_fn():
    env = make('Acrobot-v1')
    name = 'example_scheduler_'
    env_spec = EnvSpec(obs_space=env.observation_space,
                       action_space=env.action_space)

    mlp_q = MLPQValueFunction(env_spec=env_spec,
                              name_scope=name + '_mlp_q',
                              name=name + '_mlp_q',
                              mlp_config=[
                                  {
                                      "ACT": "RELU",
                                      "B_INIT_VALUE": 0.0,
                                      "NAME": "1",
                                      "N_UNITS": 16,
                                      "TYPE": "DENSE",
                                      "W_NORMAL_STDDEV": 0.03
                                  },
                                  {
                                      "ACT": "LINEAR",
                                      "B_INIT_VALUE": 0.0,
                                      "NAME": "OUPTUT",
                                      "N_UNITS": 1,
                                      "TYPE": "DENSE",
                                      "W_NORMAL_STDDEV": 0.03
                                  }
                              ])
    dqn = DQN(env_spec=env_spec,
              config_or_config_dict=dict(REPLAY_BUFFER_SIZE=1000,
                                         GAMMA=0.99,
```

(continues on next page)

(continued from previous page)

```
                                        BATCH_SIZE=10,
                                        LEARNING_RATE=0.001,
                                        TRAIN_ITERATION=1,
                                        DECAY=0.5),
              name=name + '_dqn',
              value_func=mlp_q)
    agent = Agent(env=env, env_spec=env_spec,
                  algo=dqn,
                  name=name + '_agent',
                  algo_saving_scheduler=PeriodicalEventSchedule(
                      t_fn=lambda: get_global_status_collect()('TOTAL_AGENT_TRAIN_
↪SAMPLE_COUNT'),
                      trigger_every_step=20,
                      after_t=10),
                  exploration_strategy=EpsilonGreedy(action_space=env_spec.action_
↪space,
                                                     prob_
↪scheduler=PiecewiseScheduler(
                                                         t_fn=lambda: get_global_
↪status_collect()(
                                                             'TOTAL_AGENT_TRAIN_
↪SAMPLE_COUNT'),
                                                         endpoints=((10, 0.3), (100,␣
↪0.1), (200, 0.0)),
                                                         outside_value=0.0
                                                     ),
                                                     init_random_prob=0.5))
    flow = create_train_test_flow(
        test_every_sample_count=10,
        train_every_sample_count=10,
        start_test_after_sample_count=5,
        start_train_after_sample_count=5,
        train_func_and_args=(agent.train, (), dict()),
        test_func_and_args=(agent.test, (), dict(sample_count=10)),
        sample_func_and_args=(agent.sample, (), dict(sample_count=100,
                                                     env=agent.env,
                                                     store_flag=True))
    )
    experiment = Experiment(
        tuner=None,
        env=env,
        agent=agent,
        flow=flow,
        name=name + 'experiment_debug'
    )

    dqn.parameters.set_scheduler(param_key='LEARNING_RATE',
                                 scheduler=LinearScheduler(
                                     t_fn=experiment.TOTAL_AGENT_TRAIN_SAMPLE_COUNT,
                                     schedule_timesteps=GlobalConfig().DEFAULT_
↪EXPERIMENT_END_POINT[
                                         'TOTAL_AGENT_TRAIN_SAMPLE_COUNT'],
                                     final_p=0.0001,
                                     initial_p=0.01))
    experiment.run()
```

(continues on next page)

(continued from previous page)

```
from baconian.core.experiment_runner import single_exp_runner

GlobalConfig().set('DEFAULT_LOG_PATH', './log_path')
single_exp_runner(task_fn, del_if_log_path_existed=True)
```

4.10 Visualization of experiments log

Below is the code of Baconian visualisation module. For its usage, please refer to :doc:'Logging and Visualization <how_to_log>'.

CHAPTER 5

Logging and Visualization

In this part, we will introduce how to record and save everything you want during an experiments. The contents are organized as follows:

- How to understand the log file after the experiment finished.
- How to visualize with log file after experiment finished.
- How the logging module of Baconian works.

5.1 Log Directory Explanation

After one experiment finished, lots of built-in logging information will be recorded and saved to file so you can visualize and analysis the results. A typical logging file directory looks like this:

```
.
 ├── console.log              # log output to console during experiments
 ├── final_status.json        # final status of all modules when the experiment␣
↪finished
 ├── global_config.json       # config you use for this experiment include default␣
↪config and customized config.
 ├── record                   # record folder contains all raw log of experiments.
 │    ├── agent               # agent level log (training samples' reward, test samples
↪' reward etc.)
 │    │    ├── CREATED.json    # agent's log under CREATED status
 │    │    ├── INITED.json     # agent's log after CREATED and into INITED status
 │    │    ├── TRAIN.json      # agent's log under training status (e.g., reward␣
↪received when sample for training, action during training sampling)
 │    │    └── TEST.json       # agent's log under test status (e.g., reward received␣
↪when sample for evaluation/test)
 │    └── algo                # algorithm level log (loss, KL divergence, etc.)
 │    │    ├── CREATED.json    # algorithm's log under CREATED status
 │    │    ├── INITED.json     # algorithm's log after CREATED and into INITED status
 │    │    ├── TRAIN.json      # algorithm's log under training status (e.g., training␣
↪loss)
```

(continues on next page)

(continued from previous page)

```
│   │   └── TEST.json       # algorithm's log under test status
└── model_checkpoints       # store the checkpoints of tensorflow models if you have␣
↪add the saving scheduler in your experiments.
```

5.2 Visualize and analyze the log

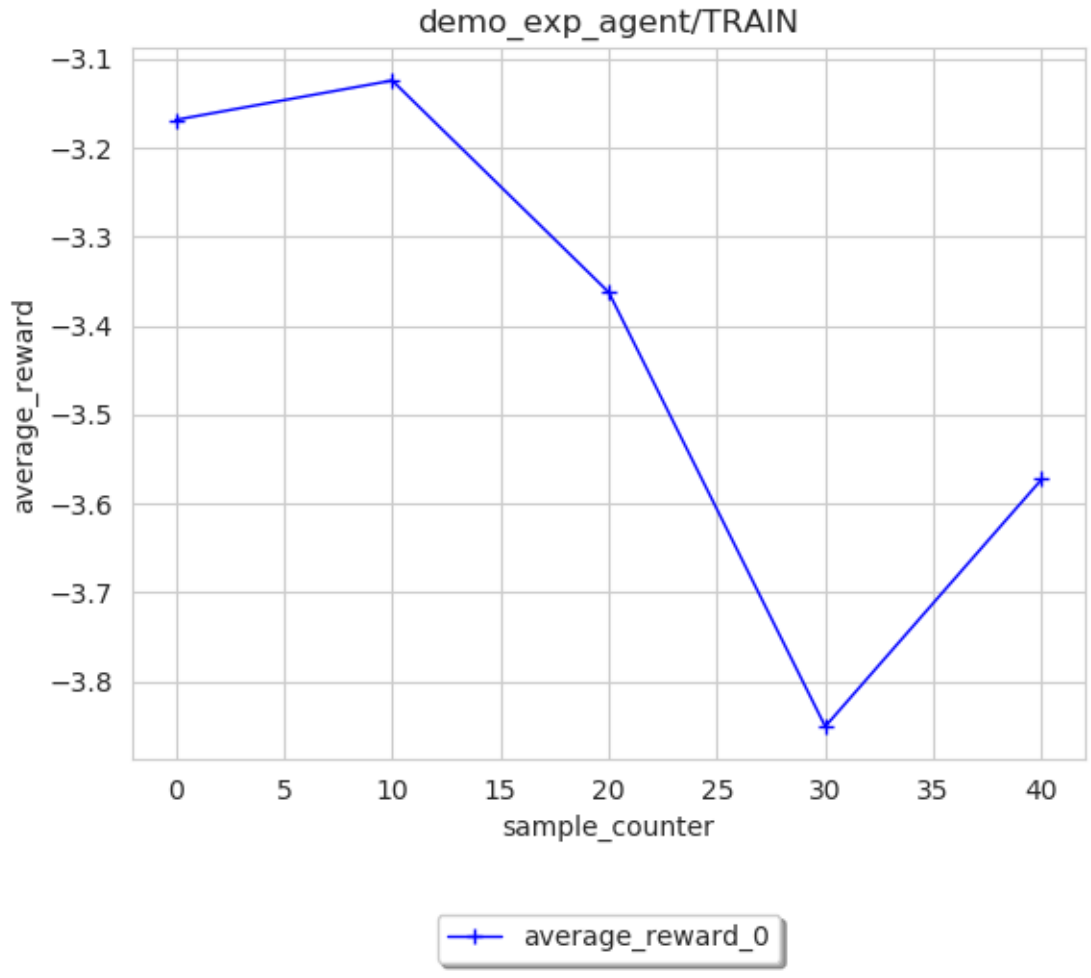
Three plot modes: `line`, `scatter` and `histogram` are available within `plot_res` function in `log_data_loader`. `plotter` will use Matplotlib package to plot a graph with `data_new` DataFrame as input. Users may utilise visualisation.py for fast plotting, with existing single or multiple experiment results.

Please refer to the following examples for single or multiple experiment visualisation:

- Single Experiment Visualisation

We use the log generated by running the *Dyna examples* as example.

Following code snippet is to draw a `line` plot of `average_reward` that agent received during training, using data sampled from environment as index and averaged every 10 rewards for smooth line.

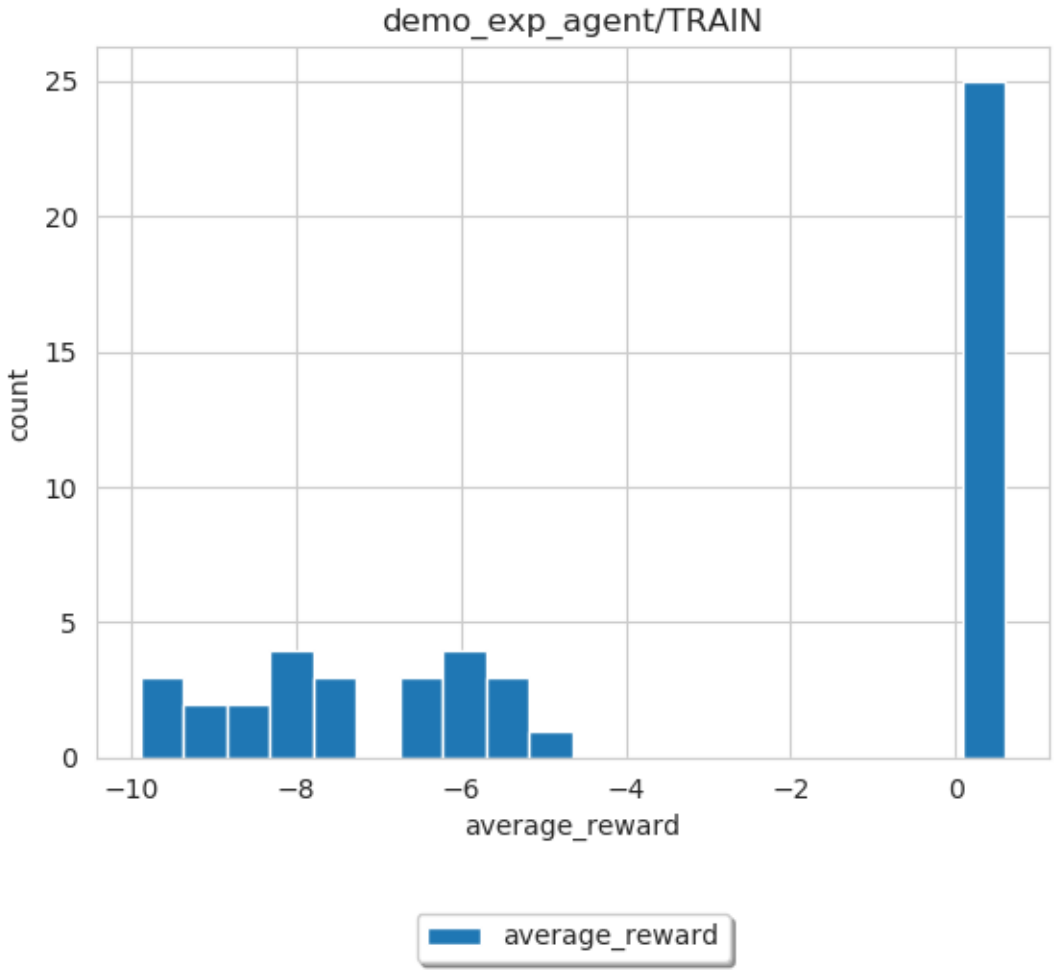


Note: `sub_log_dir_name` should include the COMPLETE directory in between the `log_path` directory and

`json.log`. For example, if you have a log folder structured as `/path/to/log/record/demo_exp_agent/TEST/log.json`, then the `sub_log_dir_name` should be `demo_exp_agent/TEST/` and `exp_root_dir` should be `/path/to/log/`.

Please note that `histogram` plot mode is a bit different from the other two modes, in terms of data manipulation. Although `index` is unnecessary under `histogram` mode, but currently user still should pass in one for internal data processing.

```
image = loader.SingleExpLogDataLoader(path)
image.plot_res(sub_log_dir_name='demo_exp_agent/TRAIN',
           key="average_reward",
           index='sample_counter',
           mode='histogram',
           save_format='pdf',
           save_flag=True,
           file_name='average_reward_histogram'
           )
```



- Multiple Experiment Visualisation

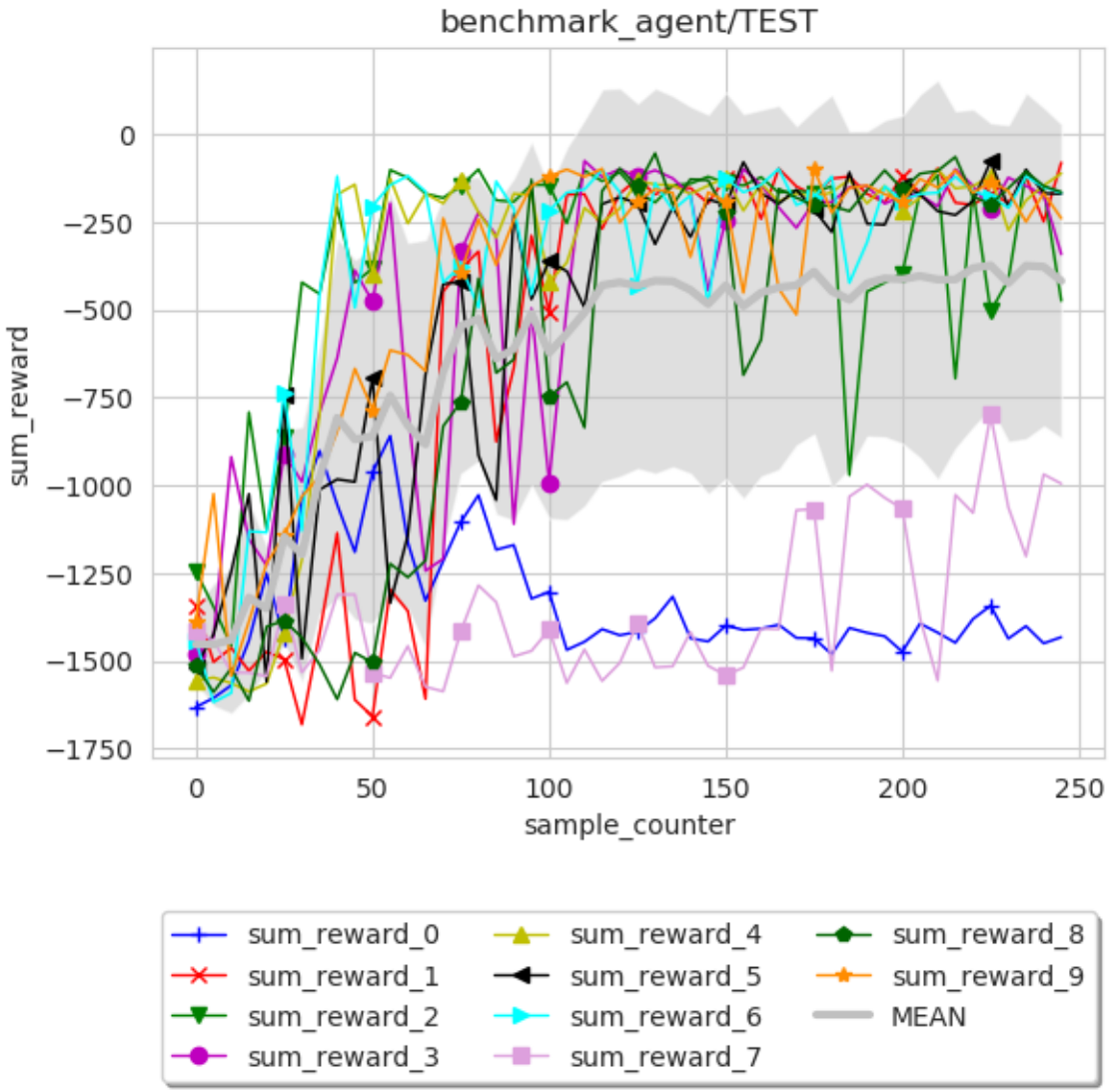
Visualize the results from multiple runs can give a more reliable analysis of the RL methods, by plotting the mean and variance over these results. Such can be done by `MultipleExpLogDataLoader`

We use the DDPG benchmark experiments as example, use can found the script under the source code

`baconian-project/baconian/benchmark/run_benchmark.py`

Following code snippet is to draw a `line` plot of `sum_reward` in `benchmark_agent/TEST` as a result of 10 times of DDPG benchmark experiments.

```
path = '/path/to/log' # under the path, there should be 10 sub folders, each contains␣
↪1 experiment results.
image = loader.MultipleExpLogDataLoader(path)
image.plot_res(sub_log_dir_name='benchmark_agent/TEST',
               key="sum_reward",
               index='sample_counter',
               mode='line',
               save_flag=True,
               average_over=10,
               save_format='pdf'
               )
```



We can see from the results that DDPG is not quite stable as 2 out of 10 runs failed to converge.

When plotting multiple experiment results in `histogram` mode, figure will reflect the histogram/data distribution using all experiments' data.

```
path = '/path/to/log'
image = loader.MultipleExpLogDataLoader(path)
image.plot_res(sub_log_dir_name='benchmark_ddpg/TRAIN',
               key="average_critic_loss",
               index='train',
               mode='histogram',
               file_name='average_critic_loss_benchmark',
               save_format='pdf',
               save_flag=True,
               )
```

We can use the action distribution to analyze and diagnose algorithms.

5.3 How the logging module of Baconian works

There are two important modules of Baconian: `Logger` and `Recorder`, `Recorder` is coupled with every module or class you want to record something during training or testing, for such as DQN, Agent or Environment. It will record the information like loss, gradient or reward in a way that you specified. While `Logger` will take charge of these recorded information, group them in a certain way and output them into file, console etc.

CHAPTER 6

How to implement a new algorithm

In this section, we will walk through the implementation of Deep Deterministic Policy Gradient (DDPG) algorithm, available at `baconian/algo/ddpg.py`. It utilizes many functionalities provided by the framework, which we will describe below.

- The `ModelFreeAlgo` and `OffPolicyAlgo` Classes

For the algorithms in Baconian project, we have writen many abstract classes to indicate the characteristics of the algorithm, in `baconian/algo/rl_algo.py`. The `DDPG` class inherits from `ModelFreeAlgo` and `OffPolicyAlgo` classes' `ModelFreeAlgo`, `OffPolicyAlgo` and other classes in `baconian/algo/rl_algo.py` inherit `Algo` class to categorize DRL algorithms.

```
from baconian.algo.algo import Algo
from baconian.algo.dynamics.dynamics_model import DynamicsModel
from baconian.core.core import EnvSpec
from baconian.common.logging import record_return_decorator
import numpy as np


class ModelFreeAlgo(Algo):
    def __init__(self, env_spec: EnvSpec, name: str = 'model_free_algo', warm_up_
↪trajectories_number=0):
        super(ModelFreeAlgo, self).__init__(env_spec, name, warm_up_trajectories_
↪number)


class OnPolicyAlgo(Algo):
    pass


class OffPolicyAlgo(Algo):
    pass


class ValueBasedAlgo(Algo):
    pass
```

(continues on next page)

(continued from previous page)

```
class PolicyBasedAlgo(Algo):
    pass


class ModelBasedAlgo(Algo):
    def __init__(self, env_spec, dynamics_model: DynamicsModel, name: str = 'model_
↪based_algo'):
        super(ModelBasedAlgo, self).__init__(env_spec, name)
        self._dynamics_model = dynamics_model
        self.dynamics_env = self._dynamics_model.return_as_env()

    def train_dynamics(self, *args, **kwargs):
        pass

    @record_return_decorator(which_recorder='self')
    def test_dynamics(self, env, sample_count, *args, **kwargs):
        self.set_status('TEST')
        env.set_status('TEST')
        st = env.reset()
        real_state_list = []
        dyanmics_state_list = []
        for i in range(sample_count):
            ac = self.env_spec.action_space.sample()
            self._dynamics_model.reset_state(state=st)
            new_state_dynamics, _, _, _ = self.dynamics_env.step(action=ac, )
            new_state_real, _, done, _ = env.step(action=ac)
            real_state_list.append(new_state_real)
            dyanmics_state_list.append(new_state_dynamics)
            st = new_state_real
            if done is True:
                env.reset()
        l1_loss = np.linalg.norm(np.array(real_state_list) - np.array(dyanmics_state_
↪list), ord=1)
        l2_loss = np.linalg.norm(np.array(real_state_list) - np.array(dyanmics_state_
↪list), ord=2)
        return dict(dynamics_test_l1_error=l1_loss, dynamics_test_l2_error=l2_loss)

    def set_terminal_reward_function_for_dynamics_env(self, terminal_func, reward_
↪func):
        self.dynamics_env.set_terminal_reward_func(terminal_func, reward_func)
```

Each new algorithm should implement the methods and attributes defined in `Algo` class (`baconian/algo/algo.py`).

```
from baconian.core.core import Basic, EnvSpec, Env
from baconian.core.status import StatusWithSubInfo
import abc
from typeguard import typechecked
from baconian.common.logging import Recorder
from baconian.core.parameters import Parameters
from baconian.common.sampler.sample_data import TrajectoryData


class Algo(Basic):
```

(continues on next page)

(continued from previous page)

```
    """
    Abstract class for algorithms
    """
    STATUS_LIST = ['CREATED', 'INITED', 'TRAIN', 'TEST']
    INIT_STATUS = 'CREATED'

    @typechecked
    def __init__(self, env_spec: EnvSpec, name: str = 'algo', warm_up_trajectories_
↪number=0):
        """
        Constructor

        :param env_spec: environment specifications
        :type env_spec: EnvSpec
        :param name: name of the algorithm
        :type name: str
        :param warm_up_trajectories_number: how many trajectories used to warm up the␣
↪training
        :type warm_up_trajectories_number: int
        """

        super().__init__(status=StatusWithSubInfo(obj=self), name=name)
        self.env_spec = env_spec
        self.parameters = Parameters(dict())
        self.recorder = Recorder(default_obj=self)
        self.warm_up_trajectories_number = warm_up_trajectories_number

    def init(self):
        """
        Initialization method, such as network random initialization in Tensorflow

        :return:
        """
        self._status.set_status('INITED')

    def warm_up(self, trajectory_data: TrajectoryData):
        """
        Use some data to warm up the algorithm, e.g., compute the mean/std-dev of the␣
↪state to perform normalization.
        Data used in warm up process will not be added into the memory
        :param trajectory_data: TrajectoryData object
        :type trajectory_data: TrajectoryData

        :return: None
        """
        pass

    def train(self, *arg, **kwargs) -> dict:
        """
        Training API, specific arguments should be defined by each algorithms itself.

        :return: training results, e.g., loss
        :rtype: dict
        """

        self._status.set_status('TRAIN')
        return dict()
```

(continues on next page)

(continued from previous page)

```
    def test(self, *arg, **kwargs) -> dict:
        """
        Testing API, most of the evaluation can be done by agent instead of␣
↪algorithms, so this API can be skipped

        :return: test results, e.g., rewards
        :rtype: dict
        """

        self._status.set_status('TEST')
        return dict()

    @abc.abstractmethod
    def predict(self, *arg, **kwargs):
        """
        Predict function, given the obs as input, return the action, obs will be read␣
↪as the first argument passed into
        this API, like algo.predict(obs=x, ...)

        :return: predicted action
        :rtype: np.ndarray
        """
        raise NotImplementedError

    @abc.abstractmethod
    def append_to_memory(self, *args, **kwargs):
        """
        For off-policy algorithm, use this API to append the data into replay buffer.␣
↪samples will be read as the first
        argument passed into this API, like algo.append_to_memory(samples=x, ...)

        """
        raise NotImplementedError

    @property
    def is_training(self):
        """
        A boolean indicate the if the algorithm is in training status

        :return: True if in training
        :rtype: bool
        """
        return self.get_status()['status'] == 'TRAIN'

    @property
    def is_testing(self):
        """
        A boolean indicate the if the algorithm is in training status

        :return: True if in testing
        :rtype: bool
        """
        return self.get_status()['status'] == 'TEST'
```

- The `MultiPlaceholderInput` Class

The algorithms in Baconian project are mostly implemented with TensorFlow, similar in the process of saving and

loading the parameters. Hence, parameters are stored in the format of TensorFlow variables by `PlaceholderInput` and `MultiPlaceholderInput` classes.

```
class DDPG(ModelFreeAlgo, OffPolicyAlgo, MultiPlaceholderInput):

    # ...

    @record_return_decorator(which_recorder='self')
    def save(self, global_step, save_path=None, name=None, **kwargs):
        save_path = save_path if save_path else GlobalConfig().DEFAULT_MODEL_
↪CHECKPOINT_PATH
        name = name if name else self.name
        MultiPlaceholderInput.save(self, save_path=save_path, global_step=global_step,
↪ name=name, **kwargs)
        return dict(check_point_save_path=save_path, check_point_save_global_
↪step=global_step,
                    check_point_save_name=name)

    @record_return_decorator(which_recorder='self')
    def load(self, path_to_model, model_name, global_step=None, **kwargs):
        MultiPlaceholderInput.load(self, path_to_model, model_name, global_step,␣
↪**kwargs)
        return dict(check_point_load_path=path_to_model, check_point_load_global_
↪step=global_step,
                    check_point_load_name=model_name)
```

- Constructor

```
class DDPG(ModelFreeAlgo, OffPolicyAlgo, MultiPlaceholderInput):
    required_key_dict = DictConfig.load_json(file_path=GlobalConfig().DEFAULT_DDPG_
↪REQUIRED_KEY_LIST)

    @typechecked()
    def __init__(self,
                 env_spec: EnvSpec,
                 config_or_config_dict: (DictConfig, dict),
                 value_func: MLPQValueFunction,
                 policy: DeterministicMLPPolicy,
                 schedule_param_list=None,
                 name='ddpg',
                 replay_buffer=None):
        """

        :param env_spec: environment specifications, like action apace or observation␣
↪space
        :param config_or_config_dict: configuraion dictionary, like learning rate or␣
↪decay, if any
        :param value_func: value function
        :param policy: agent policy
        :param schedule_param_list:
        :param name: name of algorithm class instance
        :param replay_buffer: replay buffer, if any
        """
        ModelFreeAlgo.__init__(self, env_spec=env_spec, name=name)
        config = construct_dict_config(config_or_config_dict, self)

        self.config = config
        self.actor = policy
```

(continues on next page)

(continued from previous page)

```
        self.target_actor = self.actor.make_copy(name_scope='{}_target_actor'.
↪format(self.name),
                                                 name='{}_target_actor'.format(self.
↪name),
                                                 reuse=False)
        self.critic = value_func
        self.target_critic = self.critic.make_copy(name_scope='{}_target_critic'.
↪format(self.name),
                                                   name='{}_target_critic'.
↪format(self.name),
                                                   reuse=False)

        self.state_input = self.actor.state_input

        if replay_buffer:
            assert issubclass(replay_buffer, BaseReplayBuffer)
            self.replay_buffer = replay_buffer
        else:
            self.replay_buffer = UniformRandomReplayBuffer(limit=self.config('REPLAY_
↪BUFFER_SIZE'),
                                                           action_shape=self.env_spec.
↪action_shape,
                                                           observation_shape=self.env_
↪spec.obs_shape)

        self.parameters = ParametersWithTensorflowVariable(tf_var_list=[],
                                                           rest_parameters=dict(),
                                                           to_scheduler_param_
↪tuple=schedule_param_list,
                                                           name='ddpg_param',
                                                           source_config=config,
                                                           require_snapshot=False)
        """
        self.parameters contains all the parameters (variables) of the algorithm
        """
        self._critic_with_actor_output = self.critic.make_copy(reuse=True,
                                                               name='actor_input_{}'.
↪format(self.critic.name),
                                                               state_input=self.state_
↪input,
                                                               action_input=self.
↪actor.action_tensor)
        self._target_critic_with_target_actor_output = self.target_critic.make_
↪copy(reuse=True,
                                                                                    ⌐
↪name='target_critic_with_target_actor_output_{}'.format(
                                                                                    ⌐ ⌐
↪  self.critic.name),
                                                                                    ⌐ ⌐
↪action_input=self.target_actor.action_tensor)

        with tf.variable_scope(name):
            self.reward_input = tf.placeholder(shape=[None, 1], dtype=tf.float32)
            self.next_state_input = tf.placeholder(shape=[None, self.env_spec.flat_
↪obs_dim], dtype=tf.float32)
            self.done_input = tf.placeholder(shape=[None, 1], dtype=tf.bool)
            self.target_q_input = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

(continues on next page)

(continued from previous page)

```
            done = tf.cast(self.done_input, dtype=tf.float32)
            self.predict_q_value = (1. - done) * self.config('GAMMA') * self.target_q_
↪input + self.reward_input
            with tf.variable_scope('train'):
                self.critic_loss, self.critic_update_op, self.target_critic_update_op,
↪ self.critic_optimizer, \
                self.critic_grads = self._setup_critic_loss()
                self.actor_loss, self.actor_update_op, self.target_actor_update_op,␣
↪self.action_optimizer, \
                self.actor_grads = self._set_up_actor_loss()

        var_list = get_tf_collection_var_list(
            '{}/train'.format(name)) + self.critic_optimizer.variables() + self.
↪action_optimizer.variables()
        self.parameters.set_tf_var_list(tf_var_list=sorted(list(set(var_list)),␣
↪key=lambda x: x.name))
        MultiPlaceholderInput.__init__(self,
                                       sub_placeholder_input_list=[dict(obj=self.
↪target_actor,
                                                                        attr_name=
↪'target_actor',
                                                                        ),
                                                                   dict(obj=self.
↪actor,
                                                                        attr_name=
↪'actor'),
                                                                   dict(obj=self.
↪critic,
                                                                        attr_name=
↪'critic'),
                                                                   dict(obj=self.
↪target_critic,
                                                                        attr_name=
↪'target_critic')
                                                                   ],
                                       parameters=self.parameters)
```

CHAPTER 7

How to implement a new environment

Similar to algorithms, environments in Baconian project also should implement the methods and attributes defined in `Env` class `baconian/core/core.py`, inheriting gym Env class.

```
class Env(gym.Env, Basic):
    """
    Abstract class for environment
    """
    key_list = ()
    STATUS_LIST = ('JUST_RESET', 'JUST_INITED', 'TRAIN', 'TEST', 'NOT_INIT')
    INIT_STATUS = 'NOT_INIT'

    @typechecked
    def __init__(self, name: str = 'env'):
        super(Env, self).__init__(status=StatusWithSubInfo(obj=self), name=name)
        self.action_space = None
        self.observation_space = None
        self.step_count = None
        self.recorder = Recorder()
        self._last_reset_point = 0
        self.total_step_count_fn = lambda: self._status.group_specific_info_key(info_
↪key='step', group_way='sum')

    @register_counter_info_to_status_decorator(increment=1, info_key='step', under_
↪status=('TRAIN', 'TEST'),
                                               ignore_wrong_status=True)
    def step(self, action):
        pass

    @register_counter_info_to_status_decorator(increment=1, info_key='reset', under_
↪status='JUST_RESET')
    def reset(self):
        self._status.set_status('JUST_RESET')
        self._last_reset_point = self.total_step_count_fn()
```

(continues on next page)

(continued from previous page)

```
    @register_counter_info_to_status_decorator(increment=1, info_key='init', under_
↪status='JUST_INITED')
    def init(self):
        self._status.set_status('JUST_INITED')

    def get_state(self):
        raise NotImplementedError

    def seed(self, seed=None):
        return self.unwrapped.seed(seed=seed)
```

We use STATUS to record and control the status of an environment, register_counter_info_to_status_decorator is a decorator that counts the times of initialization and reset of an environment.

```
def register_counter_info_to_status_decorator(increment, info_key, under_status: (str,
↪ tuple) = None,
                                              ignore_wrong_status=False):
    def wrap(fn):
        if under_status:
            assert isinstance(under_status, (str, tuple))
            if isinstance(under_status, str):
                final_st = tuple([under_status])
            else:
                final_st = under_status

        else:
            final_st = (None,)

        @wraps(fn)
        def wrap_with_self(self, *args, **kwargs):
            # todo record() called in fn will lost the just appended info_key at the␣
↪very first
            obj = self
            if not hasattr(obj, '_status') or not isinstance(getattr(obj, '_status'),␣
↪StatusWithInfo):
                raise ValueError(
                    ' the object {} does not not have attribute StatusWithInfo␣
↪instance or hold wrong type of Status'.format(
                        obj))

            assert isinstance(getattr(obj, '_status'), StatusWithInfo)
            obj_status = getattr(obj, '_status')
            for st in final_st:
                obj_status.append_new_info(info_key=info_key, init_value=0, under_
↪status=st)
            res = fn(self, *args, **kwargs)
            for st in final_st:
                if st and st != obj.get_status()['status'] and not ignore_wrong_
↪status:
                    raise ValueError('register counter info under status: {} but got␣
↪status {}'.format(st,
↪                 obj.get_status()[
↪                     'status']))
```

(continues on next page)

(continued from previous page)

```
            obj_status.update_info(info_key=info_key, increment=increment,
                                   under_status=obj.get_status()['status'])
            return res

        return wrap_with_self

    return wrap
```

The class `EnvSpec` stores and regulates the environment specifications, e.g. data type of observation space and action space in an environment.

```
class EnvSpec(object):
    @init_func_arg_record_decorator()
    @typechecked
    def __init__(self, obs_space: Space, action_space: Space):
        self._obs_space = obs_space
        self._action_space = action_space
        self.obs_shape = tuple(np.array(self.obs_space.sample()).shape)
        if len(self.obs_shape) == 0:
            self.obs_shape = (1,)
        self.action_shape = tuple(np.array(self.action_space.sample()).shape)
        if len(self.action_shape) == 0:
            self.action_shape = ()

    @property
    def obs_space(self):
        return self._obs_space

    @property
    def action_space(self):
        return self._action_space

    @property
    def flat_obs_dim(self) -> int:
        return int(flat_dim(self.obs_space))

    @property
    def flat_action_dim(self) -> int:
        return int(flat_dim(self.action_space))

    @staticmethod
    def flat(space: Space, obs_or_action: (np.ndarray, list)):
        return flatten(space, obs_or_action)

    def flat_action(self, action: (np.ndarray, list)):
        return flatten(self.action_space, action)

    def flat_obs(self, obs: (np.ndarray, list)):
        return flatten(self.obs_space, obs)
```

CHAPTER 8

How to implement a new dynamics model

New dynamics model in Baconian project are supposed to implement the methods and attributes defined in `DynamicsModel` class (`baconian/algo/dynamics/dynamics_model.py`).

```
    STATUS_LIST = ('CREATED', 'INITED')
    INIT_STATUS = 'CREATED'

    def __init__(self, env_spec: EnvSpec, parameters: Parameters = None, init_state=None, name='dynamics_model',
                 state_input_scaler: DataScaler = None,
                 action_input_scaler: DataScaler = None,
                 state_output_scaler: DataScaler = None):

        """

        :param env_spec: environment specifications, such as observation space and action space
        :type env_spec: EnvSpec
        :param parameters: parameters
        :type parameters: Parameters
        :param init_state: initial state of dymamics model
        :type init_state: str
        :param name: name of instance, 'dynamics_model' by default
        :type name: str
        :param state_input_scaler: data preprocessing scaler of state input
        :type state_input_scaler: DataScaler
        :param action_input_scaler: data preprocessing scaler of action input
        :type action_input_scaler: DataScaler
        :param state_output_scaler: data preprocessing scaler of state output
        :type state_output_scaler: DataScaler
        """
        super().__init__(name=name)
        self.env_spec = env_spec
        self.state = init_state
        self.parameters = parameters
```

(continues on next page)

(continued from previous page)

```
        self.state_input = None
        self.action_input = None
        self.new_state_output = None
        self.recorder = Recorder(flush_by_split_status=False, default_obj=self)
        self._status = StatusWithSingleInfo(obj=self)
        self.state_input_scaler = state_input_scaler if state_input_scaler else IdenticalDataScaler(
            dims=env_spec.flat_obs_dim)
        self.action_input_scaler = action_input_scaler if action_input_scaler else IdenticalDataScaler(
            dims=env_spec.flat_action_dim)
        self.state_output_scaler = state_output_scaler if state_output_scaler else IdenticalDataScaler(
            dims=env_spec.flat_obs_dim)

    def init(self, *args, **kwargs):
        self.set_status('INITED')
        self.state = self.env_spec.obs_space.sample()

    @register_counter_info_to_status_decorator(increment=1, info_key='step_counter')
    def step(self, action: np.ndarray, state=None, allow_clip=False, **kwargs_for_transit):

        """
        State transition function (only support one sample transition instead of batch data)

        :param action: action to be taken
        :type action: np.ndarray
        :param state: current state, if None, will use stored state (saved from last transition)
        :type state: np.ndarray
        :param allow_clip: allow clip of observation space, default False
        :type allow_clip: bool
        :param kwargs_for_transit: extra kwargs for calling the _state_transit, this is typically related to the
                                   specific mode you used
        :type kwargs_for_transit:
        :return: new state after step
        :rtype: np.ndarray
        """
        state = np.array(state).reshape(self.env_spec.obs_shape) if state is not None else self.state
        action = action.reshape(self.env_spec.action_shape)
        if allow_clip is True:
            if state is not None:
                    state = self.env_spec.obs_space.clip(state)
            action = self.env_spec.action_space.clip(action)
        if self.env_spec.action_space.contains(action) is False:
            raise StateOrActionOutOfBoundError(
                'action {} out of bound of {}'.format(action, self.env_spec.action_space.bound()))
        if self.env_spec.obs_space.contains(state) is False:
            raise StateOrActionOutOfBoundError(
                'state {} out of bound of {}'.format(state, self.env_spec.obs_space.bound()))
        new_state = self._state_transit(state=state, action=self.env_spec.flat_action(action),
```

(continues on next page)

(continued from previous page)

```
                                          **kwargs_for_transit)
        if allow_clip is True:
            new_state = self.env_spec.obs_space.clip(new_state)
        if self.env_spec.obs_space.contains(new_state) is False:
            raise StateOrActionOutOfBoundError(
                'new state {} out of bound of {}'.format(new_state, self.env_spec.obs_
↪space.bound()))
        self.state = new_state
        return new_state

    @abc.abstractmethod
    def _state_transit(self, state, action, **kwargs) -> np.ndarray:
        """

        :param state: original state
        :type state: np.ndarray
        :param action:  action taken by agent
        :type action: np.ndarray
        :param kwargs:
        :type kwargs:
        :return: new state after transition
        :rtype: np.ndarray
        """
        raise NotImplementedError

    def copy_from(self, obj) -> bool:
        """

        :param obj: object to copy from
        :type obj:
        :return: True if successful else raise an error
        :rtype: bool
        """
        if not isinstance(obj, type(self)):
            raise TypeError('Wrong type of obj %s to be copied, which should be %s' %␣
↪(type(obj), type(self)))
        return True

    def make_copy(self):
        """ Make a copy of parameters and environment specifications."""
        raise NotImplementedError

    def reset_state(self, state=None):
        """

        :param state: original state
        :type state: np.ndarray
        :return: a random sample space in observation space
        :rtype: np.ndarray
        """
        if state is not None:
            assert self.env_spec.obs_space.contains(state)
            self.state = state
        else:
            self.state = self.env_spec.obs_space.sample()

    def return_as_env(self) -> Env:
```

(continues on next page)

(continued from previous page)

```
        """

        :return: an environment with this dynamics model
        :rtype: DynamicsEnvWrapper
        """
        return DynamicsEnvWrapper(dynamics=self,
                                  name=self._name + '_env')
```

Similar to algorithms, dynamics models are categorized in `baconian/algo/dynamics/dynamics_model.py`, such as `GlobalDynamicsModel` and `DifferentiableDynamics`.

CHAPTER 9

System Overview of Baconian

The system overview of Baconian is shown in below figure. We design Baconian with the objective to minimize users' coding effort on developing and testing MBRL algorithms. With Baconian, the user can easily setup a MBRL experiment by configuring the target algorithms and modules without the need for understanding the inner mechanisms.

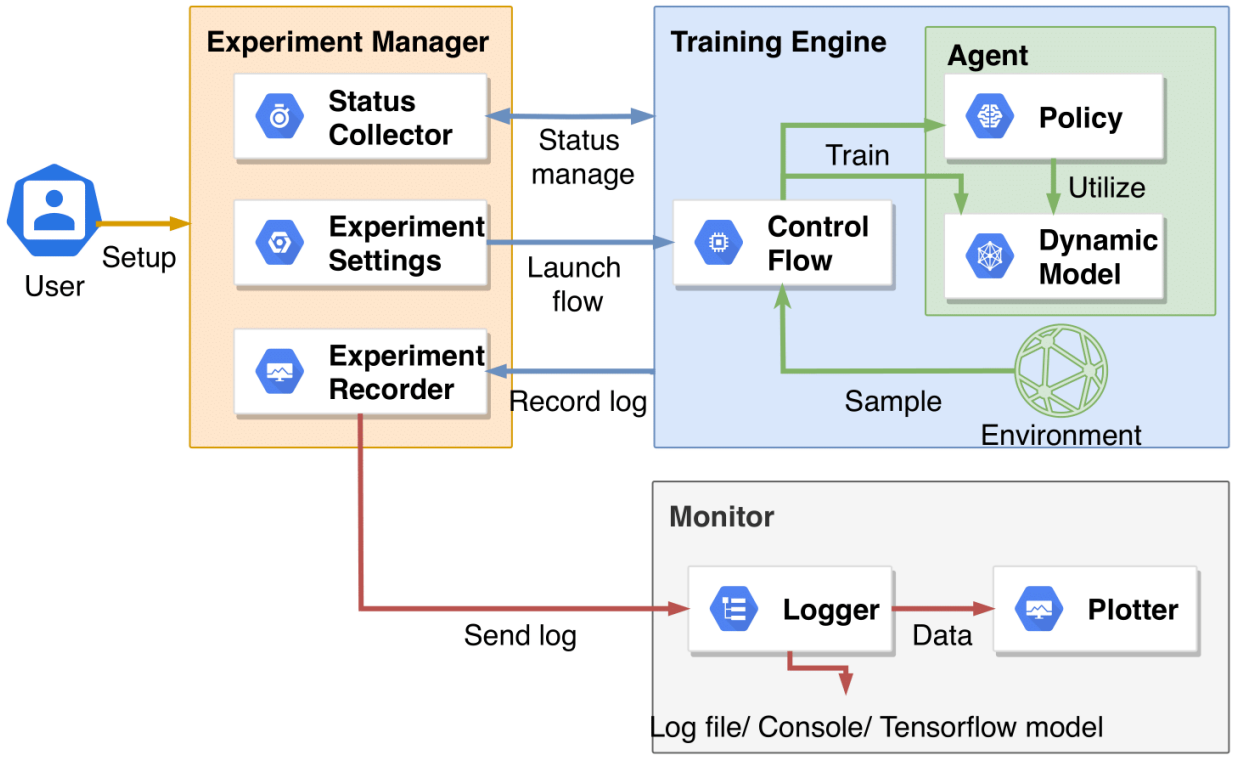


Fig. 1: System Overview of Baconian.

Following figure shows the experiment creation flow of Baconian.

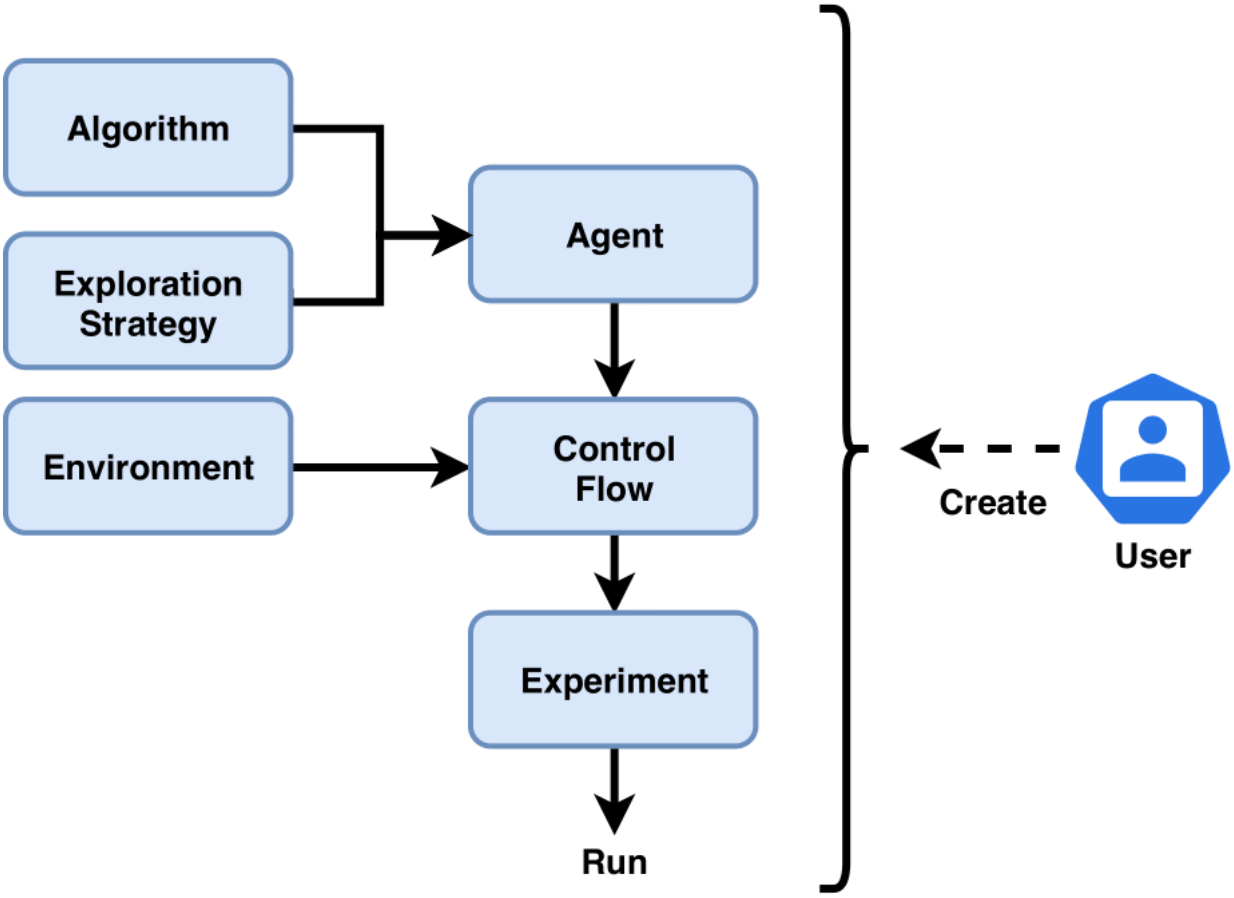


Fig. 2: Baconian experiment creation process.

First, the user should create a environment and a RL algorithm module with necessary hyper-parameters configured, e.g., neural network size, learning rate. Algorithm module is usually composed by a policy module and a dynamics model module depending on different algorithms. Then, the user needs to create an agent module by passing the algorithm module, and the exploration strategy module if needed, into it. Second, the user should create a control flow module that defines how the experiments should be proceeded and the stopping conditions. This includes defining how much samples should be collected for training at each step, and what condition to indicate the completion of an experiment, etc. Some typical control flows have already been implemented in Baconian to meet the users' needs.

Finally, an experiment module should be created by passing the agent, environment, control flow modules into it, and then launched. After that, the Baconian will handle the experiment running, monitoring, and results saving/logging, etc.

For more information on system design of Baconian, you can read our demo paper at: https://arxiv.org/abs/1904.10762.

CHAPTER 10

Baconian API Reference

10.1 Core Module

10.1.1 baconian.core.core.Basic

class baconian.core.core.**Basic**(*name: str*, *status=None*)

Basic class within the whole framework

INIT_STATUS = None

STATUS_LIST = ('TRAIN', 'TEST')

__init__(*name: str*, *status=None*)

Init a new Basic instance.

Parameters

- **name** (*str*) – name of the object, can be determined to generate log path, handle tensorflow name scope etc.
- **status** (*Status*) – A status instance Status to indicate the status of the object

allow_duplicate_name = False

get_status() → dict

Return the object's status, a dictionary.

init(**args*, ***kwargs*)

Initialize the object

load(**args*, ***kwargs*)

Load the parameters from training checkpoints.

name

The name(id) of object, a string.

required_key_dict = ()

save(**args*, ***kwargs*)
Save the parameters in training checkpoints.

set_status(*val*)
Set the object's status.

status_list
Status list of the object, ('TRAIN', 'TEST').

10.1.2 baconian.core.core.Env

class baconian.core.core.**Env**(*name: str = 'env'*, *copy_from_env=None*)
Abstract class for environment

INIT_STATUS = 'CREATED'

STATUS_LIST = ('JUST_RESET', 'INITED', 'TRAIN', 'TEST', 'CREATED')

__init__(*name: str = 'env'*, *copy_from_env=None*)

get_state()
Get the status of the environment.

init()
Set the status to 'INITED'.

key_list = ()

reset()
Set the status to 'JUST_RESET', and update new reset point

seed(*seed=None*)

Parameters **seed** (*int*) – seed to generate random number

Returns seed of the unwrapped environment

Return type int

step(*action*)

Parameters **action** (*method*) – agent's action, the environment will react responding to action

10.1.3 baconian.core.core.EnvSpec

class baconian.core.core.**EnvSpec**(*obs_space: baconian.common.spaces.base.Space*, *action_space: baconian.common.spaces.base.Space*)

__init__(*obs_space: baconian.common.spaces.base.Space*, *action_space: baconian.common.spaces.base.Space*)

action_shape

action_space

Returns Action space of environment

Return type Space

static flat(*space: baconian.common.spaces.base.Space*, *obs_or_action: (<class 'numpy.ndarray'>, <class 'list'>)*)

flat the input obs or action :param space: space of environment :type space: Space :param obs_or_action: action or observation space :type obs_or_action: (np.ndarray, list) :return: flatten action or observation space :rtype: Space

flat_action(*action: (<class 'numpy.ndarray'>, <class 'list'>)*)

Parameters **action** (*(np.ndarray, list)*) – action taken by agent

Returns flatten action parameter

Return type np.ndarray

flat_action_dim

Returns the dimension(length) of flatten action space

Return type int

flat_obs(*obs: (<class 'numpy.ndarray'>, <class 'list'>)*)

Parameters **obs** (*(np.ndarray, list)*) – observation of the agent

Returns flatten observation parameter

Return type np.ndarray

flat_obs_dim

Returns the dimension(length) of flatten observation space

Return type int

obs_shape

obs_space

Returns Observation space of environment

Return type Space

10.2 Agent module

10.2.1 baconian.core.agent

class baconian.core.agent.**Agent**(*name*, *env: (<class 'baconian.core.core.Env'>, <class 'baconian.envs.env_wrapper.Wrapper'>)*, *algo: baconian.algo.algo.Algo*, *env_spec: baconian.core.core.EnvSpec*, *sampler: baconian.common.sampler.sampler.Sampler = None*, *noise_adder: baconian.common.noise.AgentActionNoiseWrapper = None*, *reset_noise_every_terminal_state=False*, *reset_state_every_sample=False*, *exploration_strategy: baconian.algo.misc.epsilon_greedy.ExplorationStrategy = None*, *algo_saving_scheduler: baconian.common.schedules.EventScheduler = None*)

INIT_STATUS = 'CREATED'

STATUS_LIST = ('CREATED', 'INITED', 'TRAIN', 'TEST')

__init__(*name*, *env: (<class 'baconian.core.core.Env'>, <class 'baconian.envs.env_wrapper.Wrapper'>)*, *algo: baconian.algo.algo.Algo*, *env_spec: baconian.core.core.EnvSpec*, *sampler: baconian.common.sampler.sampler.Sampler = None*, *noise_adder: baconian.common.noise.AgentActionNoiseWrapper = None*, *reset_noise_every_terminal_state=False*, *reset_state_every_sample=False*, *exploration_strategy: baconian.algo.misc.epsilon_greedy.ExplorationStrategy = None*, *algo_saving_scheduler: baconian.common.schedules.EventScheduler = None*)

Parameters

- **name** (*str*) – the name of the agent instance
- **env** (Env) – environment that interacts with agent
- **algo** (Algo) – algorithm of the agent
- **env_spec** (EnvSpec) – environment specifications: action apace and environment space
- **sampler** (Sampler) – sampler
- **reset_noise_every_terminal_state** (*bool*) – reset the noise every sampled trajectory
- **reset_state_every_sample** (*bool*) – reset the state everytime perofrm the sample/rollout
- **noise_adder** (*AgentActionNoiseWrapper*) – add action noise for exploration in action space
- **exploration_strategy** (*ExplorationStrategy*) – exploration strategy in action space
- **algo_saving_scheduler** (*EventSchedule*) – control the schedule the varying parameters in training process

init()

Initialize the algorithm, and set status to 'INITED'.

is_testing

Check whether the agent is testing. Return a boolean value.

Returns true if the agent is testing

Return type bool

is_training

Check whether the agent is training. Return a boolean value.

Returns true if the agent is training

Return type bool

predict(***kwargs*)

predict the action given the state

Parameters **kwargs** – rest parameters, include key: obs

Returns predicted action

Return type numpy ndarray

required_key_dict = {}

reset_on_terminal_state()

sample(*env*, *sample_count: int*, *in_which_status: str = 'TRAIN'*, *store_flag=False*, *sample_type: str = 'transition'*) -> (<class 'baconian.common.sampler.sample_data.TransitionData'>, <class 'baconian.common.sampler.sample_data.TrajectoryData'>)
sample a certain number of data from environment

Parameters

- **env** – environment to sample
- **sample_count** – int, sample count
- **in_which_status** – string, environment status
- **store_flag** – to store environment samples or not, default False
- **sample_type** – the type of sample, 'transition' by default

Returns sample data from environment

Return type some subclass of SampleData: TrajectoryData or TransitionData

store_samples(*samples: baconian.common.sampler.sample_data.SampleData*)
store the samples into memory/replay buffer if the algorithm that agent hold need to do so, like DQN, DDPG

Parameters **samples** (*SampleData*) – sample data of the experiment

test(*sample_count*) → baconian.common.sampler.sample_data.SampleData
test the agent

Parameters **sample_count** (*int*) – how many trajectories used to evaluate the agent's performance

Returns SampleData object.

train(**args*, ***kwargs*)
train the agent

Returns True for successfully train the agent, false if memory buffer did not have enough data.

Return type bool

10.3 Experiment modules

10.3.1 baconian.core.experiment

For experiments, its functionality should include: 1. experiment and config set up 2. logging control 3. hyper-param tuning etc. 4. visualization 5. any related experiment utility ...

class baconian.core.experiment.**Experiment**(*name: str*, *agent: baconian.core.agent.Agent*, *env: baconian.core.core.Env*, *flow: baconian.core.flow.train_test_flow.Flow*, *tuner: baconian.core.tuner.Tuner = None*, *register_default_global_status=True*)

INIT_STATUS = 'CREATED'

STATUS_LIST = ('CREATED', 'INITED', 'RUNNING', 'FINISHED', 'CORRUPTED')

TOTAL_AGENT_TEST_SAMPLE_COUNT()

TOTAL_AGENT_TRAIN_SAMPLE_COUNT()

TOTAL_AGENT_UPDATE_COUNT()

TOTAL_ENV_STEP_TEST_SAMPLE_COUNT()

TOTAL_ENV_STEP_TRAIN_SAMPLE_COUNT()

__init__(*name: str*, *agent: baconian.core.agent.Agent*, *env: baconian.core.core.Env*, *flow: baconian.core.flow.train_test_flow.Flow*, *tuner: baconian.core.tuner.Tuner = None*, *register_default_global_status=True*)

Parameters

- **name** (*str*) – name of experiment
- **agent** (Agent) – agent of experiment
- **env** (Env) – environment of experiment
- **flow** (Flow) – control flow to experiment
- **tuner** (*Tuner*) – hyper-parameter tuning method, currently in development
- **register_default_global_status** (*bool*) – register info key and status into global status collection

init()
Create a new TensorFlow session, and set status to 'INITED'.

required_key_dict = {}

run()
Run the experiment, and set status to 'RUNNING'.

10.3.2 baconian.core.experiment_runner

baconian.core.experiment_runner.**single_exp_runner**(*task_fn*, *auto_choose_gpu_flag=False*, *gpu_id: int = 0*, *seed=None*, *del_if_log_path_existed=False*, ***task_fn_kwargs*)

Parameters

- **task_fn** (*method*) – task function defined bu users
- **auto_choose_gpu_flag** (*bool*) – auto choose gpu, default False
- **gpu_id** (*int*) – gpu id, default 0
- **seed** (*int*) – seed generated by system time

:param del_if_log_path_existed:delete obsolete log file path if existed, by default False :type del_if_log_path_existed: bool :param task_fn_kwargs: :type task_fn_kwargs: :return: :rtype:

baconian.core.experiment_runner.**duplicate_exp_runner**(*num*, *task_fn*, *auto_choose_gpu_flag=False*, *gpu_id: int = 0*, *seeds: list = None*, *del_if_log_path_existed=False*, ***task_fn_kwargs*)

Parameters

- **num** (*int*) – the number of multiple experiments

- **task_fn** (*method*) – task function, defined by users
- **auto_choose_gpu_flag** (*bool*) – auto choose gpu, default False
- **gpu_id** (*int*) – gpu id, default 0
- **seeds** (*list*) – seeds generated by system time
- **del_if_log_path_existed** (*bool*) – delete the existing log path, default False
- **task_fn_kwargs** –

Returns

Return type

10.4 Environment Module

10.4.1 baconian.envs.gym_env

Gym environment wrapping module

baconian.envs.gym_env.GymEnv.**name**
The name(id) of object, a string.

baconian.envs.gym_env.GymEnv.**status_list**
Status list of the object, ('TRAIN', 'TEST').

baconian.envs.gym_env.GymEnv.**unwrapped**

Returns original unwrapped gym environment

Return type gym env

baconian.envs.gym_env.GymEnv.**unwrapped_gym**

Returns gym environment, depend on attribute 'unwrapped'

Return type gym env

10.5 Dynamics Module

10.5.1 baconian.algo.dynamics.dynamics_model.DynamicsModel

class baconian.algo.dynamics.dynamics_model.**DynamicsModel**(*env_spec: baconian.core.core.EnvSpec*, *parameters: baconian.core.parameters.Parameters = None*, *init_state=None*, *name='dynamics_model'*, *state_input_scaler: baconian.common.data_pre_processing.DataScaler = None*, *action_input_scaler: baconian.common.data_pre_processing.DataScaler = None*, *state_output_scaler: baconian.common.data_pre_processing.DataScaler = None*)

INIT_STATUS = 'CREATED'

STATUS_LIST = ('CREATED', 'INITED')

__init__(*env_spec: baconian.core.core.EnvSpec*, *parameters: baconian.core.parameters.Parameters = None*, *init_state=None*, *name='dynamics_model'*, *state_input_scaler: baconian.common.data_pre_processing.DataScaler = None*, *action_input_scaler: baconian.common.data_pre_processing.DataScaler = None*, *state_output_scaler: baconian.common.data_pre_processing.DataScaler = None*)

Parameters

- **env_spec** (EnvSpec) – environment specifications, such as observation space and action space
- **parameters** (*Parameters*) – parameters
- **init_state** (*str*) – initial state of dymamics model
- **name** (*str*) – name of instance, 'dynamics_model' by default
- **state_input_scaler** (*DataScaler*) – data preprocessing scaler of state input
- **action_input_scaler** (*DataScaler*) – data preprocessing scaler of action input
- **state_output_scaler** (*DataScaler*) – data preprocessing scaler of state output

copy_from(*obj*) → bool

Parameters **obj** – object to copy from

Returns True if successful else raise an error

Return type bool

init(**args*, ***kwargs*)

make_copy()
Make a copy of parameters and environment specifications.

reset_state(*state=None*)

Parameters **state** (*np.ndarray*) – original state

Returns a random sample space in observation space

Return type np.ndarray

return_as_env() → baconian.core.core.Env

Returns an environment with this dynamics model

Return type DynamicsEnvWrapper

step(*action: numpy.ndarray*, *state=None*, *allow_clip=False*, ***kwargs_for_transit*)
State transition function (only support one sample transition instead of batch data)

Parameters

- **action** (*np.ndarray*) – action to be taken
- **state** (*np.ndarray*) – current state, if None, will use stored state (saved from last transition)
- **allow_clip** (*bool*) – allow clip of observation space, default False
- **kwargs_for_transit** – extra kwargs for calling the _state_transit, this is typically related to the specific mode you used

Returns new state after step

Return type np.ndarray

10.5.2 Abstraction class for different types of dynamics model:

class baconian.algo.dynamics.dynamics_model.**LocalDyanmicsModel**(*env_spec: baconian.core.core.EnvSpec*, *parameters: baconian.core.parameters.Parameters = None*, *init_state=None*, *name='dynamics_model'*, *state_input_scaler: baconian.common.data_pre_processing.Data = None*, *action_input_scaler: baconian.common.data_pre_processing.Data = None*, *state_output_scaler: baconian.common.data_pre_processing.Data = None*)

class baconian.algo.dynamics.dynamics_model.**GlobalDynamicsModel**(*env_spec: baconian.core.core.EnvSpec*, *parameters: baconian.core.parameters.Parameters = None*, *init_state=None*, *name='dynamics_model'*, *state_input_scaler: baconian.common.data_pre_processing.Dat = None*, *action_input_scaler: baconian.common.data_pre_processing.Dat = None*, *state_output_scaler: baconian.common.data_pre_processing.Dat = None*)

class baconian.algo.dynamics.dynamics_model.**TrainableDyanmicsModel**

train(**args*, ***kwargs*)

class baconian.algo.dynamics.dynamics_model.**DifferentiableDynamics**(*input_node_dict: dict*, *output_node_dict: dict*)

__init__(*input_node_dict: dict*, *output_node_dict: dict*)

grad_on_input(*key_or_node: (<class 'str'>, <sphinx.ext.autodoc.importer._MockObject object at 0x7fda1ca940b8>)*, *order=1*, *batch_flag=False*)

split_and_hessian(*out_node*, *innode*)

10.6 Algorithm Module

10.6.1 baconian.algo.algo.Algo

class baconian.algo.algo.**Algo**(*env_spec: baconian.core.core.EnvSpec*, *name: str = 'algo'*, *warm_up_trajectories_number=0*)

Abstract class for algorithms

INIT_STATUS = 'CREATED'

STATUS_LIST = ['CREATED', 'INITED', 'TRAIN', 'TEST']

__init__(*env_spec: baconian.core.core.EnvSpec*, *name: str = 'algo'*, *warm_up_trajectories_number=0*)

Constructor

Parameters

- **env_spec** (EnvSpec) – environment specifications

- **name** (*str*) – name of the algorithm
- **warm_up_trajectories_number** (*int*) – how many trajectories used to warm up the training

append_to_memory(**args*, ***kwargs*)
For off-policy algorithm, use this API to append the data into replay buffer. samples will be read as the first argument passed into this API, like algo.append_to_memory(samples=x, ...)

init()
Initialization method, such as network random initialization in Tensorflow

Returns

is_testing
A boolean indicate the if the algorithm is in training status

Returns True if in testing

Return type bool

is_training
A boolean indicate the if the algorithm is in training status

Returns True if in training

Return type bool

predict(**arg*, ***kwargs*)
Predict function, given the obs as input, return the action, obs will be read as the first argument passed into this API, like algo.predict(obs=x, ...)

Returns predicted action

Return type np.ndarray

test(**arg*, ***kwargs*) → dict
Testing API, most of the evaluation can be done by agent instead of algorithms, so this API can be skipped

Returns test results, e.g., rewards

Return type dict

train(**arg*, ***kwargs*) → dict
Training API, specific arguments should be defined by each algorithms itself.

Returns training results, e.g., loss

Return type dict

warm_up(*trajectory_data: baconian.common.sampler.sample_data.TrajectoryData*)
Use some data to warm up the algorithm, e.g., compute the mean/std-dev of the state to perform normalization. Data used in warm up process will not be added into the memory :param trajectory_data: TrajectoryData object :type trajectory_data: TrajectoryData

Returns None

10.7 Flow Module

10.7.1 baconian.core.flow.train_test_flow.Flow

class baconian.core.flow.train_test_flow.**Flow**(*func_dict*)

Interface of experiment flow module, it defines the workflow of the reinforcement learning experiments.

__init__(*func_dict*)

Constructor for Flow.

Parameters **func_dict** (*dict*) – the function and its arguments that will be called in the Flow

_call_func(*key*, ***extra_kwargs*)

Call a function that is pre-defined in self.func_dict

Parameters

- **key** (*str*) – name of the function, e.g., train, test, sample.
- **extra_kwargs** – some extra kwargs you may want to be passed in the function calling

Returns actual return value of the called function if self.func_dict has such function otherwise None.

Return type

_launch() → bool

Abstract method to be implemented by subclass for a certain workflow.

Returns True if the flow correctly executed and finished

Return type bool

launch() → bool

Launch the flow until it finished or catch a system-allowed errors (e.g., out of GPU memory, to ensure the log will be saved safely).

Returns True if the flow correctly executed and finished

Return type bool

required_func = ()

required_key_dict = {}

class baconian.core.flow.train_test_flow.**TrainTestFlow**(*train_sample_count_func*, *config_or_config_dict: (<class 'baconian.config.dict_config.DictConfig'>, <class 'dict'>)*, *func_dict: dict*)

A typical sampling-trainning and testing workflow, that used by most of model-free/model-based reinforcement learning method. Typically, it repeat the sampling(saving to memory if off policy)->training(from memory if off-policy, from samples if on-policy)->test

__init__(*train_sample_count_func*, *config_or_config_dict: (<class 'baconian.config.dict_config.DictConfig'>, <class 'dict'>)*, *func_dict: dict*)

Constructor of TrainTestFlow

Parameters

- **train_sample_count_func** (*method*) – a function indicates how much training samples the agent has collected currently.
- **config_or_config_dict** (*Config or dict*) – a Config or a dict should have the keys: (TEST_EVERY_SAMPLE_COUNT, TRAIN_EVERY_SAMPLE_COUNT, START_TRAIN_AFTER_SAMPLE_COUNT, START_TEST_AFTER_SAMPLE_COUNT)
- **func_dict** (*dict*) – function dict, holds the keys: 'sample', 'train', 'test'. each item in the dict as also should be a dict, holds the keys 'func', 'args', 'kwargs'

_is_ended()

Returns True if an experiment is ended

Return type bool

_launch() → bool

Launch the flow until it finished or catch a system-allowed errors (e.g., out of GPU memory, to ensure the log will be saved safely).

Returns True if the flow correctly executed and finished

Return type bool

required_func = ('train', 'test', 'sample')

required_key_dict = {'START_TEST_AFTER_SAMPLE_COUNT': 1, 'START_TRAIN_AFTER_SAMPLE_COU

10.7.2 baconian.core.flow.dyna_flow.DynaFlow

class baconian.core.flow.dyna_flow.**DynaFlow**(*train_sample_count_func*, *config_or_config_dict: (<class 'baconian.config.dict_config.DictConfig'>, <class 'dict'>)*, *func_dict: dict*)

A typical flow for utilizing the model-based algo, it is not restricted to Dyna algorithms, but can be utilized by others.

__init__(*train_sample_count_func*, *config_or_config_dict: (<class 'baconian.config.dict_config.DictConfig'>, <class 'dict'>)*, *func_dict: dict*)

Parameters

- **train_sample_count_func** (*method*) – a function indicates how much training samples the agent has collected currently.
- **config_or_config_dict** (*Config or dict*) – a Config or a dict should have the keys: (TEST_EVERY_SAMPLE_COUNT, TRAIN_EVERY_SAMPLE_COUNT, START_TRAIN_AFTER_SAMPLE_COUNT, START_TEST_AFTER_SAMPLE_COUNT)
- **func_dict** (*dict*) – function dict, holds the keys: 'sample', 'train', 'test'. each item in the dict as also should be a dict, holds the keys 'func', 'args', 'kwargs'

_is_ended()

Returns True if an experiment is ended

Return type bool

_launch() → bool

Launch the flow until it finished or catch a system-allowed errors (e.g., out of GPU memory, to ensure the log will be saved safely).

Returns True if the flow correctly executed and finished

Return type bool

required_func = ('train_algo', 'train_algo_from_synthesized_data', 'train_dynamics', '

required_key_dict = {'START_TEST_ALGO_AFTER_SAMPLE_COUNT': 1, 'START_TEST_DYNAMICS_AFT

10.8 Common module

10.8.1 baconian.common

class baconian.common.sampler.sampler.**Sampler**

Sampler module that handle the sampling procedure for training/testing of the agent.

static sample(*env: baconian.core.core.Env*, *agent*, *sample_count: int*, *sample_type='transition'*, *reset_at_start=None*) -> (*<class 'baconian.common.sampler.sample_data.TransitionData'>*, *<class 'baconian.common.sampler.sample_data.TrajectoryData'>*)

a static method of sample function

Parameters

- **env** – environment object to sample from.
- **agent** – agent object to offer the sampling policy
- **in_which_status** – a string, "TEST" or "TRAIN" indicate this sample is used for training or testing (evaluation)
- **sample_count** – number of samples. If the sample_type == "transition", then this value means the number of transitions, usually for off-policy method like DQN, DDPG. If the sample_type == "trajectory", then this value means the numbers of trajectories.
- **sample_type** – a string, "transition" or "trajectory".
- **reset_at_start** – A bool, if True, will reset the environment at the beginning, if False, continue sampling based on previous state (this is useful for certain tasks that you need to preserve previous state to reach the terminal goal state). If None, for sample_type == "transition", it will set to False, for sample_type == "trajectory", it will set to True.

Returns SampleData object.

10.9 Visualisation Module

10.9.1 baconian.common.plotter

class baconian.common.plotter.**Plotter**

color_list = ['b', 'r', 'g', 'm', 'y', 'k', 'cyan', 'plum', 'darkgreen', 'darkorange',

markers = ('+', 'x', 'v', 'o', '^', '<', '>', 's', 'p', '*', 'h', 'H', 'D', 'd', 'P',

static plot_any_key_in_log(*data*, *key*, *index*, *exp_num=1*, *sub_log_dir_name=None*, *scatter_flag=False*, *histogram_flag=False*, *save_flag=False*, *save_path=None*, *save_format='png'*, *file_name=None*, *separate_exp_flag=True*, *mean_stddev_flag=False*, *path_list=None*)

Parameters

- **data** – a pandas DataFrame containing (index and) key columns
- **key** – in y-axis, the variable to plot, assigned by user
- **index** – in x-axis, the argument assigned by user
- **sub_log_dir_name** – the sub-directory which the log file to plot is saved in
- **exp_num** – [optional] the number of experiments to visualize
- **scatter_flag** – [optional] draw scatter plot if true
- **histogram_flag** – [optional] draw histogram if true
- **save_flag** – [optional] save the figure to a file if true
- **save_path** – [optional] save path of figure, assigned by user, the directory of log_path by default
- **save_format** – [optional] format of figure to save, png by default
- **file_name** – [optional] the file name of the file to save, key_VERUS_index by default
- **separate_exp_flag** – [optional] plot the results of each experiment separately if true
- **mean_stddev_flag** – [optional] plot the mean value of multiple experiment results and standard deviation
- **path_list** – [optional] the list of save paths assigned by users, figure file will be saved to each path

Returns

static plot_any_scatter_in_log(*res_dict*, *res_name*, *file_name*, *key*, *index*, *op*, *scatter_flag=False*, *save_flag=False*, *save_path=None*, *fig_id=4*, *label=''*, *restrict_dict=None*)

plot_fig(*fig_num*, *col_id*, *x*, *y*, *title*, *x_label*, *y_label*, *label=' '*, *marker='*'*)

CHAPTER 11

Contributions

We hope everyone that is working on Model-Based RL research can contribute to this project to better boost the development of Model-Based RL research.

Please submit an issue in the GitHub repository if you run into any problem or bug. Also feel free to contact me: linsen001@e.ntu.edu.sg.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index

Symbols

A

B

C

D

E

F

S

T

U

W