Tianshou

Release 0.5.0

Tianshou contributors

Mar 13, 2023

TUTORIALS

Tianshou () is a reinforcement learning platform based on pure PyTorch. Unlike existing reinforcement learning libraries, which are mainly based on TensorFlow, have many nested classes, unfriendly API, or slow-speed, Tianshou provides a fast-speed framework and pythonic API for building the deep reinforcement learning agent. The supported interface algorithms include:

- *DQNPolicy* Deep Q-Network
- *DQNPolicy* Double DQN
- *DQNPolicy* Dueling DQN
- *BranchingDQNPolicy* Branching DQN
- *C51Policy* Categorical DQN
- *RainbowPolicy* Rainbow DQN
- *QRDQNPolicy* Quantile Regression DQN
- *IQNPolicy* Implicit Quantile Network
- *FQFPolicy* Fully-parameterized Quantile Function
- *PGPolicy* Policy Gradient
- *NPGPolicy* Natural Policy Gradient
- *A2CPolicy* Advantage Actor-Critic
- *TRPOPolicy* Trust Region Policy Optimization
- *PPOPolicy* Proximal Policy Optimization
- *DDPGPolicy* Deep Deterministic Policy Gradient
- *TD3Policy* Twin Delayed DDPG
- *SACPolicy* Soft Actor-Critic
- *REDQPolicy* Randomized Ensembled Double Q-Learning
- *DiscreteSACPolicy* Discrete Soft Actor-Critic
- *ImitationPolicy* Imitation Learning
- *BCQPolicy* Batch-Constrained deep Q-Learning
- *CQLPolicy* Conservative Q-Learning
- *TD3BCPolicy* Twin Delayed DDPG with Behavior Cloning
- *DiscreteBCQPolicy* Discrete Batch-Constrained deep Q-Learning
- *DiscreteCQLPolicy* Discrete Conservative Q-Learning
- *DiscreteCRRPolicy* Critic Regularized Regression
- *GAILPolicy* Generative Adversarial Imitation Learning
- *PSRLPolicy* Posterior Sampling Reinforcement Learning
- *ICMPolicy* Intrinsic Curiosity Module
- *PrioritizedReplayBuffer* Prioritized Experience Replay
- *compute_episodic_return()* Generalized Advantage Estimator
- *HERReplayBuffer* Hindsight Experience Replay

Here is Tianshou's other features:

- Elegant framework, using only ~3000 lines of code
- State-of-the-art MuJoCo benchmark
- Support vectorized environment (synchronous or asynchronous) for all algorithms: *Parallel Sampling*
- Support super-fast vectorized environment EnvPool for all algorithms: *EnvPool Integration*
- Support recurrent state representation in actor network and critic network (RNN-style training for POMDP): *RNN-style Training*
- Support any type of environment state/action (e.g. a dict, a self-defined class, ...): *User-defined Environment and Different State Representation*
- Support *Customize Training Process*
- Support n-step returns estimation `compute_nstep_return()` and prioritized experience replay `PrioritizedReplayBuffer` for all Q-learning based algorithms; GAE, nstep and PER are very fast thanks to numba jit function and vectorized numpy operation
- Support *Multi-Agent RL*
- Support both TensorBoard and W&B log tools
- Support multi-GPU training *Multi-GPU Training*
- Comprehensive unit tests, including functional checking, RL pipeline checking, documentation checking, PEP8 code-style checking, and type checking

https://tianshou.readthedocs.io/zh/master/

CHAPTER

ONE

INSTALLATION

Tianshou is currently hosted on PyPI and conda-forge. It requires Python >= 3.6.

You can simply install Tianshou from PyPI with the following command:

```
$ pip install tianshou
```

If you use Anaconda or Miniconda, you can install Tianshou from conda-forge through the following command:

```
$ conda -c conda-forge install tianshou
```

You can also install with the newest version through GitHub:

```
$ pip install git+https://github.com/thu-ml/tianshou.git@master --upgrade
```

After installation, open your python console and type

```
import tianshou
print(tianshou.__version__)
```

If no error occurs, you have successfully installed Tianshou.

Tianshou is still under development, you can also check out the documents in stable version through tianshou.readthedocs.io/en/stable/.

1.1 Get Started with Jupyter Notebook

In this tutorial, we will use Google Colaboratory to show you the most basic usages of common building blocks in Tianshou. You will be guided step by step to see how different modules in Tianshou collaborate with each other to conduct a classic DRL experiment (PPO algorithm for CartPole-v0 environment).

- L0: Overview
- L1: Batch
- L2: Replay Buffer
- L3: Vectorized Environment
- L4: Policy
- L5: Collector
- L6: Trainer
- L7: Experiment

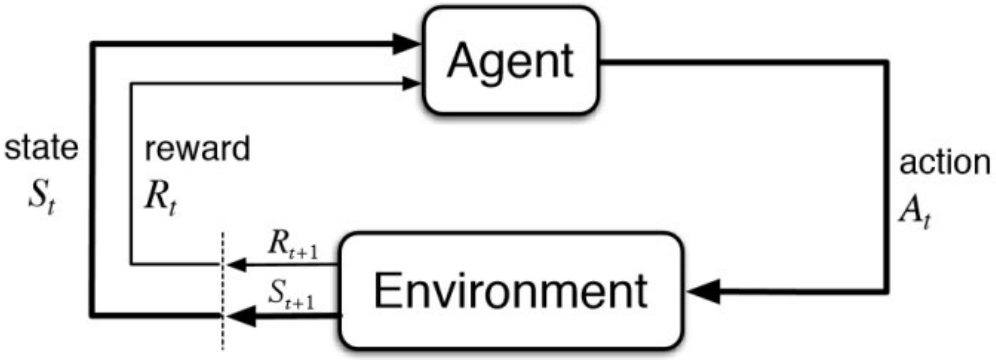
1.2 Deep Q Network

Deep reinforcement learning has achieved significant successes in various applications. **Deep Q Network** (DQN) [MKS+15] is the pioneer one. In this tutorial, we will show how to train a DQN agent on CartPole with Tianshou step by step. The full script is at test/discrete/test_dqn.py.

Contrary to existing Deep RL libraries such as RLlib, which could only accept a config specification of hyperparameters, network, and others, Tianshou provides an easy way of construction through the code-level.

1.2.1 Overview

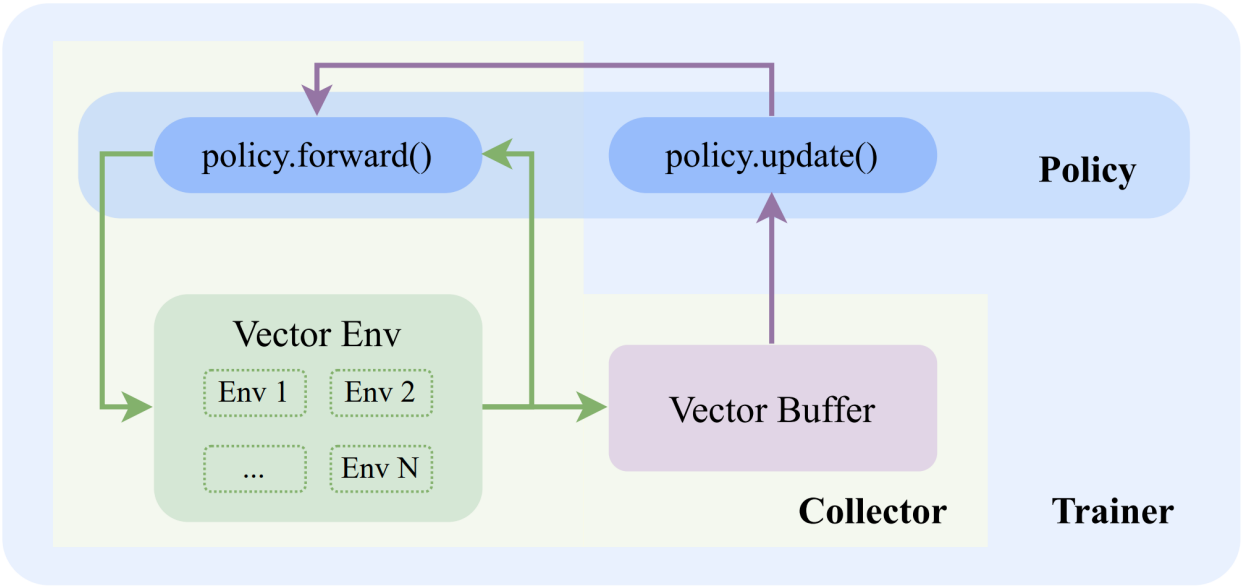
In reinforcement learning, the agent interacts with environments to improve itself.



There are three types of data flow in RL training pipeline:

1. Agent to environment: `action` will be generated by agent and sent to environment;
2. Environment to agent: `env.step` takes action, and returns a tuple of `(observation, reward, done, info)`;
3. Agent-environment interaction to agent training: the data generated by interaction will be stored and sent to the learner of agent.

In the following sections, we will set up (vectorized) environments, policy (with neural network), collector (with buffer), and trainer to successfully run the RL training and evaluation pipeline. Here is the overall system:



1.2.2 Make an Environment

First of all, you have to make an environment for your agent to interact with. You can use `gym.make(environment_name)` to make an environment for your agent. For environment interfaces, we follow the convention of Gymnasium. In your Python code, simply import Tianshou and make the environment:

```
import gymnasium as gym
import tianshou as ts

env = gym.make('CartPole-v0')
```

CartPole-v0 includes a cart carrying a pole moving on a track. This is a simple environment with a discrete action space, for which DQN applies. You have to identify whether the action space is continuous or discrete and apply eligible algorithms. DDPG [LHP+16], for example, could only be applied to continuous action spaces, while almost all other policy gradient methods could be applied to both.

Here is the detail of useful fields of CartPole-v0:

- `state`: the position of the cart, the velocity of the cart, the angle of the pole and the velocity of the tip of the pole;
- `action`: can only be one of `[0, 1, 2]`, for moving the cart left, no move, and right;
- `reward`: each timestep you last, you will receive a +1 `reward`;
- `done`: if CartPole is out-of-range or timeout (the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center, or you last over 200 timesteps);
- `info`: extra info from environment simulation.

The goal is to train a good policy that can get the highest reward in this environment.

1.2.3 Setup Vectorized Environment

If you want to use the original `gym.Env`:

```
train_envs = gym.make('CartPole-v0')
test_envs = gym.make('CartPole-v0')
```

Tianshou supports vectorized environment for all algorithms. It provides four types of vectorized environment wrapper:

- *DummyVectorEnv*: the sequential version, using a single-thread for-loop;
- *SubprocVectorEnv*: use python multiprocessing and pipe for concurrent execution;
- *ShmemVectorEnv*: use share memory instead of pipe based on SubprocVectorEnv;
- *RayVectorEnv*: use Ray for concurrent activities and is currently the only choice for parallel simulation in a cluster with multiple machines. It can be used as follows: (more explanation can be found at *Parallel Sampling*)

```
train_envs = ts.env.DummyVectorEnv([lambda: gym.make('CartPole-v0') for _ in range(10)])
test_envs = ts.env.DummyVectorEnv([lambda: gym.make('CartPole-v0') for _ in range(100)])
```

Here, we set up 10 environments in `train_envs` and 100 environments in `test_envs`.

You can also try the super-fast vectorized environment EnvPool by

```
import envpool
train_envs = envpool.make_gymnasium("CartPole-v0", num_envs=10)
test_envs = envpool.make_gymnasium("CartPole-v0", num_envs=100)
```

For the demonstration, here we use the second code-block.

Warning: If you use your own environment, please make sure the `seed` method is set up properly, e.g.,

```
def seed(self, seed):
    np.random.seed(seed)
```

Otherwise, the outputs of these envs may be the same with each other.

1.2.4 Build the Network

Tianshou supports any user-defined PyTorch networks and optimizers. Yet, of course, the inputs and outputs must comply with Tianshou's API. Here is an example:

```
import torch, numpy as np
from torch import nn

class Net(nn.Module):
    def __init__(self, state_shape, action_shape):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(np.prod(state_shape), 128), nn.ReLU(inplace=True),
            nn.Linear(128, 128), nn.ReLU(inplace=True),
            nn.Linear(128, 128), nn.ReLU(inplace=True),
            nn.Linear(128, np.prod(action_shape)),
        )
```

(continues on next page)

(continued from previous page)

```
    def forward(self, obs, state=None, info={}):
        if not isinstance(obs, torch.Tensor):
            obs = torch.tensor(obs, dtype=torch.float)
        batch = obs.shape[0]
        logits = self.model(obs.view(batch, -1))
        return logits, state

state_shape = env.observation_space.shape or env.observation_space.n
action_shape = env.action_space.shape or env.action_space.n
net = Net(state_shape, action_shape)
optim = torch.optim.Adam(net.parameters(), lr=1e-3)
```

You can also use pre-defined MLP networks in *common*, *discrete*, and *continuous*. The rules of self-defined networks are:

1. Input: observation `obs` (may be a `numpy.ndarray`, `torch.Tensor`, dict, or self-defined class), hidden state `state` (for RNN usage), and other information `info` provided by the environment.
2. Output: some `logits`, the next hidden state `state`. The logits could be a tuple instead of a `torch.Tensor`, or some other useful variables or results during the policy forwarding procedure. It depends on how the policy class process the network output. For example, in PPO [SWD+17], the return of the network might be `(mu, sigma), state` for Gaussian policy.

Note: The logits here indicates the raw output of the network. In supervised learning, the raw output of prediction/classification model is called logits, and here we extend this definition to any raw output of the neural network.

1.2.5 Setup Policy

We use the defined `net` and `optim` above, with extra policy hyper-parameters, to define a policy. Here we define a DQN policy with a target network:

```
policy = ts.policy.DQNPolicy(net, optim, discount_factor=0.9, estimation_step=3, target_
↪update_freq=320)
```

1.2.6 Setup Collector

The collector is a key concept in Tianshou. It allows the policy to interact with different types of environments conveniently. In each step, the collector will let the policy perform (at least) a specified number of steps or episodes and store the data in a replay buffer.

The following code shows how to set up a collector in practice. It is worth noticing that VectorReplayBuffer is to be used in vectorized environment scenarios, and the number of buffers, in the following case 10, is preferred to be set as the number of environments.

```
train_collector = ts.data.Collector(policy, train_envs, ts.data.VectorReplayBuffer(20000,
↪ 10), exploration_noise=True)
test_collector = ts.data.Collector(policy, test_envs, exploration_noise=True)
```

The main function of collector is the collect function, which can be summarized in the following lines:

```
result = self.policy(self.data, last_state)                          # the agent predicts␣
↪the batch action from batch observation
act = to_numpy(result.act)
self.data.update(act=act)                                            # update the data␣
↪with new action/policy
result = self.env.step(act, ready_env_ids)                           # apply action to␣
↪environment
obs_next, rew, done, info = result
self.data.update(obs_next=obs_next, rew=rew, done=done, info=info)  # update the data␣
↪with new state/reward/done/info
```

1.2.7 Train Policy with a Trainer

Tianshou provides *onpolicy_trainer()*, *offpolicy_trainer()*, and *offline_trainer()*. The trainer will automatically stop training when the policy reach the stop condition `stop_fn` on test collector. Since DQN is an off-policy algorithm, we use the *offpolicy_trainer()* as follows:

```
result = ts.trainer.offpolicy_trainer(
    policy, train_collector, test_collector,
    max_epoch=10, step_per_epoch=10000, step_per_collect=10,
    update_per_step=0.1, episode_per_test=100, batch_size=64,
    train_fn=lambda epoch, env_step: policy.set_eps(0.1),
    test_fn=lambda epoch, env_step: policy.set_eps(0.05),
    stop_fn=lambda mean_rewards: mean_rewards >= env.spec.reward_threshold)
print(f'Finished training! Use {result["duration"]}')
```

The meaning of each parameter is as follows (full description can be found at *offpolicy_trainer()*):

- `max_epoch`: The maximum of epochs for training. The training process might be finished before reaching the `max_epoch`;
- `step_per_epoch`: The number of environment step (a.k.a. transition) collected per epoch;
- `step_per_collect`: The number of transition the collector would collect before the network update. For example, the code above means "collect 10 transitions and do one policy network update";
- `episode_per_test`: The number of episodes for one policy evaluation.
- `batch_size`: The batch size of sample data, which is going to feed in the policy network.
- `train_fn`: A function receives the current number of epoch and step index, and performs some operations at the beginning of training in this epoch. For example, the code above means "reset the epsilon to 0.1 in DQN before training".
- `test_fn`: A function receives the current number of epoch and step index, and performs some operations at the beginning of testing in this epoch. For example, the code above means "reset the epsilon to 0.05 in DQN before testing".
- `stop_fn`: A function receives the average undiscounted returns of the testing result, return a boolean which indicates whether reaching the goal.
- `logger`: See below.

The trainer supports TensorBoard for logging. It can be used as:

```
from torch.utils.tensorboard import SummaryWriter
from tianshou.utils import TensorboardLogger
writer = SummaryWriter('log/dqn')
logger = TensorboardLogger(writer)
```

Pass the logger into the trainer, and the training result will be recorded into the TensorBoard.

The returned result is a dictionary as follows:

```
{
    'train_step': 9246,
    'train_episode': 504.0,
    'train_time/collector': '0.65s',
    'train_time/model': '1.97s',
    'train_speed': '3518.79 step/s',
    'test_step': 49112,
    'test_episode': 400.0,
    'test_time': '1.38s',
    'test_speed': '35600.52 step/s',
    'best_reward': 199.03,
    'duration': '4.01s'
}
```

It shows that within approximately 4 seconds, we finished training a DQN agent on CartPole. The mean returns over 100 consecutive episodes is 199.03.

1.2.8 Save/Load Policy

Since the policy inherits the class `torch.nn.Module`, saving and loading the policy are exactly the same as a torch module:

```
torch.save(policy.state_dict(), 'dqn.pth')
policy.load_state_dict(torch.load('dqn.pth'))
```

1.2.9 Watch the Agent's Performance

Collector supports rendering. Here is the example of watching the agent's performance in 35 FPS:

```
policy.eval()
policy.set_eps(0.05)
collector = ts.data.Collector(policy, env, exploration_noise=True)
collector.collect(n_episode=1, render=1 / 35)
```

If you'd like to manually see the action generated by a well-trained agent:

```
# assume obs is a single environment observation
action = policy(Batch(obs=np.array([obs]))).act[0]
```

1.2.10 Train a Policy with Customized Codes

"I don't want to use your provided trainer. I want to customize it!"

Tianshou supports user-defined training code. Here is the code snippet:

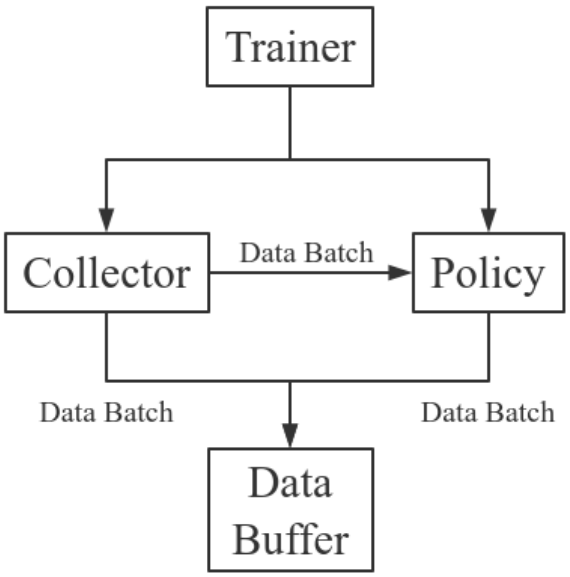
```
# pre-collect at least 5000 transitions with random action before training
train_collector.collect(n_step=5000, random=True)

policy.set_eps(0.1)
for i in range(int(1e6)):  # total step
    collect_result = train_collector.collect(n_step=10)

    # once if the collected episodes' mean returns reach the threshold,
    # or every 1000 steps, we test it on test_collector
    if collect_result['rews'].mean() >= env.spec.reward_threshold or i % 1000 == 0:
        policy.set_eps(0.05)
        result = test_collector.collect(n_episode=100)
        if result['rews'].mean() >= env.spec.reward_threshold:
            print(f'Finished training! Test mean returns: {result["rews"].mean()}')
            break
        else:
            # back to training eps
            policy.set_eps(0.1)

    # train policy with a sampled batch data from buffer
    losses = policy.update(64, train_collector.buffer)
```

For further usage, you can refer to the *Cheat Sheet*.

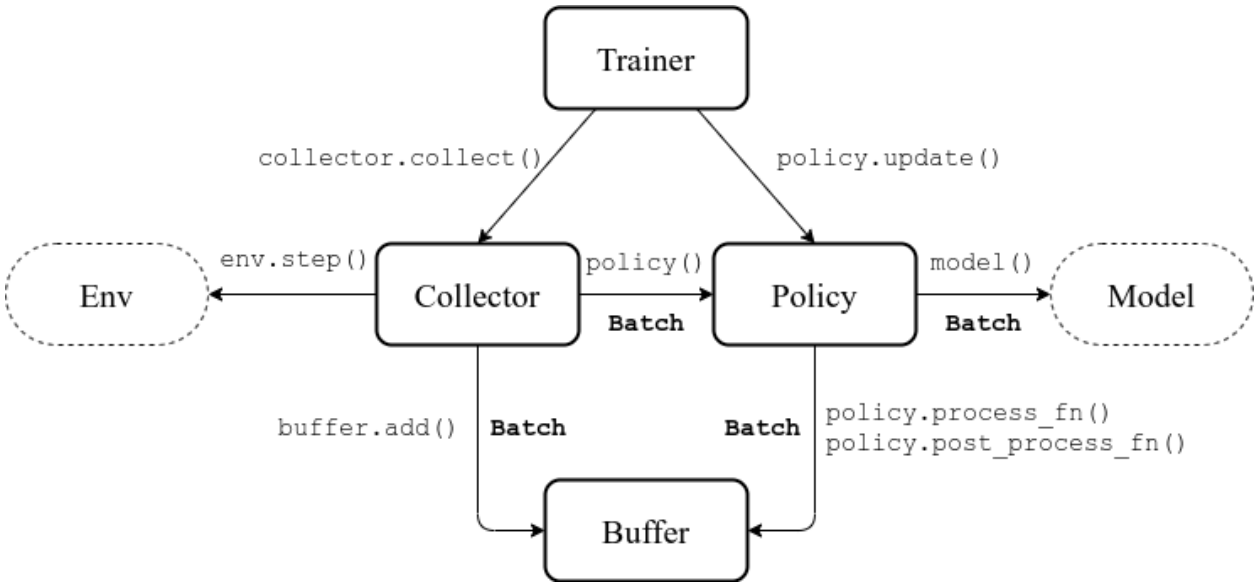
References

1.3 Basic concepts in Tianshou

Tianshou splits a Reinforcement Learning agent training procedure into these parts: trainer, collector, policy, and data buffer. The general control flow can be described as:



Here is a more detailed description, where `Env` is the environment and `Model` is the neural network:



1.3.1 Batch

Tianshou provides *Batch* as the internal data structure to pass any kind of data to other methods, for example, a collector gives a *Batch* to policy for learning. Let's take a look at this script:

```
>>> import torch, numpy as np
>>> from tianshou.data import Batch
>>> data = Batch(a=4, b=[5, 5], c='2312312', d=('a', -2, -3))
>>> # the list will automatically be converted to numpy array
>>> data.b
array([5, 5])
>>> data.b = np.array([3, 4, 5])
```

(continues on next page)

(continued from previous page)

```
>>> print(data)
Batch(
    a: 4,
    b: array([3, 4, 5]),
    c: '2312312',
    d: array(['a', '-2', '-3'], dtype=object),
)
>>> data = Batch(obs={'index': np.zeros((2, 3))}, act=torch.zeros((2, 2)))
>>> data[:, 1] += 6
>>> print(data[-1])
Batch(
    obs: Batch(
             index: array([0., 6., 0.]),
         ),
    act: tensor([0., 6.]),
)
```

In short, you can define a *Batch* with any key-value pair, and perform some common operations over it.

Understand Batch is a dedicated tutorial for *Batch*. We strongly recommend every user to read it so as to correctly understand and use *Batch*.

1.3.2 Buffer

ReplayBuffer stores data generated from interaction between the policy and environment. ReplayBuffer can be considered as a specialized form (or management) of *Batch*. It stores all the data in a batch with circular-queue style.

The current implementation of Tianshou typically use the following reserved keys in *Batch*:

- `obs` the observation of step t ;
- `act` the action of step t ;
- `rew` the reward of step t ;
- `terminated` the terminated flag of step t ;
- `truncated` the truncated flag of step t ;
- `done` the done flag of step t (can be inferred as `terminated or truncated`);
- `obs_next` the observation of step $t+1$;
- `info` the info of step t (in `gym.Env`, the `env.step()` function returns 4 arguments, and the last one is `info`);
- `policy` the data computed by policy in step t;

When adding data to a replay buffer, the done flag will be inferred automatically from `terminated``and ``truncated`.

The following code snippet illustrates the usage, including:

- the basic data storage: `add()`;
- get attribute, get slicing data, ...;
- sample from buffer: `sample_indices(batch_size)` and `sample(batch_size)`;
- get previous/next transition index within episodes: `prev(index)` and `next(index)`;
- save/load data from buffer: pickle and HDF5;

```
>>> import pickle, numpy as np
>>> from tianshou.data import Batch, ReplayBuffer
>>> buf = ReplayBuffer(size=20)
>>> for i in range(3):
...     buf.add(Batch(obs=i, act=i, rew=i, terminated=0, truncated=0, obs_next=i + 1,␣
↪info={}))

>>> buf.obs
# since we set size = 20, len(buf.obs) == 20.
array([0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>> # but there are only three valid items, so len(buf) == 3.
>>> len(buf)
3
>>> # save to file "buf.pkl"
>>> pickle.dump(buf, open('buf.pkl', 'wb'))
>>> # save to HDF5 file
>>> buf.save_hdf5('buf.hdf5')

>>> buf2 = ReplayBuffer(size=10)
>>> for i in range(15):
...     terminated = i % 4 == 0
...     buf2.add(Batch(obs=i, act=i, rew=i, terminated=terminated, truncated=False, obs_
↪next=i + 1, info={}))
>>> len(buf2)
10
>>> buf2.obs
# since its size = 10, it only stores the last 10 steps' result.
array([10, 11, 12, 13, 14,  5,  6,  7,  8,  9])

>>> # move buf2's result into buf (meanwhile keep it chronologically)
>>> buf.update(buf2)
>>> buf.obs
array([ 0,  1,  2,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14,  0,  0,  0,
        0,  0,  0,  0,  0])

>>> # get all available index by using batch_size = 0
>>> indices = buf.sample_indices(0)
>>> indices
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
>>> # get one step previous/next transition
>>> buf.prev(indices)
array([ 0,  0,  1,  2,  3,  4,  5,  7,  7,  8,  9, 11, 11])
>>> buf.next(indices)
array([ 1,  2,  3,  4,  5,  6,  6,  8,  9, 10, 10, 12, 12])

>>> # get a random sample from buffer
>>> # the batch_data is equal to buf[indices].
>>> batch_data, indices = buf.sample(batch_size=4)
>>> batch_data.obs == buf[indices].obs
array([ True,  True,  True,  True])
>>> len(buf)
13
```

(continues on next page)

(continued from previous page)

```
>>> buf = pickle.load(open('buf.pkl', 'rb'))  # load from "buf.pkl"
>>> len(buf)
3
>>> # load complete buffer from HDF5 file
>>> buf = ReplayBuffer.load_hdf5('buf.hdf5')
>>> len(buf)
3
```

ReplayBuffer also supports frame_stack sampling (typically for RNN usage, see issue#19), ignoring storing the next observation (save memory in Atari tasks), and multi-modal observation (see issue#38):

```
>>> buf = ReplayBuffer(size=9, stack_num=4, ignore_obs_next=True)
>>> for i in range(16):
...     terminated = i % 5 == 0
...     ptr, ep_rew, ep_len, ep_idx = buf.add(
...         Batch(obs={'id': i}, act=i, rew=i,
...               terminated=terminated, truncated=False, obs_next={'id': i + 1}))
...     print(i, ep_len, ep_rew)
0 [1] [0.]
1 [0] [0.]
2 [0] [0.]
3 [0] [0.]
4 [0] [0.]
5 [5] [15.]
6 [0] [0.]
7 [0] [0.]
8 [0] [0.]
9 [0] [0.]
10 [5] [40.]
11 [0] [0.]
12 [0] [0.]
13 [0] [0.]
14 [0] [0.]
15 [5] [65.]
>>> print(buf)  # you can see obs_next is not saved in buf
ReplayBuffer(
    obs: Batch(
             id: array([ 9, 10, 11, 12, 13, 14, 15,  7,  8]),
         ),
    act: array([ 9, 10, 11, 12, 13, 14, 15,  7,  8]),
    rew: array([ 9., 10., 11., 12., 13., 14., 15.,  7.,  8.]),
    done: array([False, True, False, False, False, False, True, False,
                 False]),
)
>>> index = np.arange(len(buf))
>>> print(buf.get(index, 'obs').id)
[[ 7  7  8  9]
 [ 7  8  9 10]
 [11 11 11 11]
 [11 11 11 12]
 [11 11 12 13]
 [11 12 13 14]
```

(continues on next page)

(continued from previous page)

```
 [12 13 14 15]
 [ 7  7  7  7]
 [ 7  7  7  8]]
>>> # here is another way to get the stacked data
>>> # (stack only for obs and obs_next)
>>> abs(buf.get(index, 'obs')['id'] - buf[index].obs.id).sum().sum()
0
>>> # we can get obs_next through __getitem__, even if it doesn't exist
>>> # however, [:] will select the item according to timestamp,
>>> # that equals to index == [7, 8, 0, 1, 2, 3, 4, 5, 6]
>>> print(buf[:].obs_next.id)
[[ 7  7  7  8]
 [ 7  7  8  9]
 [ 7  8  9 10]
 [ 7  8  9 10]
 [11 11 11 12]
 [11 11 12 13]
 [11 12 13 14]
 [12 13 14 15]
 [12 13 14 15]]
>>> full_index = np.array([7, 8, 0, 1, 2, 3, 4, 5, 6])
>>> np.allclose(buf[:].obs_next.id, buf[full_index].obs_next.id)
True
```

Tianshou provides other type of data buffer such as *PrioritizedReplayBuffer* (based on Segment Tree and `numpy.ndarray`) and *VectorReplayBuffer* (add different episodes' data but without losing chronological order). Check out *ReplayBuffer* for more detail.

1.3.3 Policy

Tianshou aims to modularize RL algorithms. It comes into several classes of policies in Tianshou. All of the policy classes must inherit *BasePolicy*.

A policy class typically has the following parts:

- `__init__()`: initialize the policy, including copying the target network and so on;
- *forward()*: compute action with given observation;
- *process_fn()*: pre-process data from the replay buffer;
- *learn()*: update policy with a given batch of data.
- *post_process_fn()*: update the buffer with a given batch of data.
- *update()*: the main interface for training. This function samples data from buffer, pre-process data (such as computing n-step return), learn with the data, and finally post-process the data (such as updating prioritized replay buffer); in short, `process_fn -> learn -> post_process_fn`.

States for policy

During the training process, the policy has two main states: training state and testing state. The training state can be further divided into the collecting state and updating state.

The meaning of training and testing state is obvious: the agent interacts with environment, collects training data and performs update, that's training state; the testing state is to evaluate the performance of the current policy during training process.

As for the collecting state, it is defined as interacting with environments and collecting training data into the buffer; we define the updating state as performing a model update by *update()* during training process.

In order to distinguish these states, you can check the policy state by `policy.training` and `policy.updating`. The state setting is as follows:

State for policy		policy.training	policy.updating
Training state	Collecting state	True	False
	Updating state	True	True
Testing state		False	False

`policy.updating` is helpful to distinguish the different exploration state, for example, in DQN we don't have to use epsilon-greedy in a pure network update, so `policy.updating` is helpful for setting epsilon in this case.

policy.forward

The `forward` function computes the action over given observations. The input and output is algorithm-specific but generally, the function is a mapping of `(batch, state, ...) -> batch`.

The input batch is the environment data (e.g., observation, reward, done flag and info). It comes from either *collect()* or *sample()*. The first dimension of all variables in the input `batch` should be equal to the batch-size.

The output is also a Batch which must contain "act" (action) and may contain "state" (hidden state of policy), "policy" (the intermediate result of policy which needs to save into the buffer, see *forward()*), and some other algorithm-specific keys.

For example, if you try to use your policy to evaluate one episode (and don't want to use *collect()*), use the following code-snippet:

```
# assume env is a gym.Env
obs, done = env.reset(), False
while not done:
    batch = Batch(obs=[obs])  # the first dimension is batch-size
    act = policy(batch).act[0]  # policy.forward return a batch, use ".act" to extract␣
↪the action
    obs, rew, done, info = env.step(act)
```

Here, `Batch(obs=[obs])` will automatically create the 0-dimension to be the batch-size. Otherwise, the network cannot determine the batch-size.

policy.process_fn

The `process_fn` function computes some variables that depends on time-series. For example, compute the N-step or GAE returns.

Take 2-step return DQN as an example. The 2-step return DQN compute each transition's return as:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 \max_a Q(s_{t+2}, a)$$

where γ is the discount factor, $\gamma \in [0, 1]$. Here is the pseudocode showing the training process **without Tianshou framework**:

```
# pseudocode, cannot work
obs = env.reset()
buffer = Buffer(size=10000)
agent = DQN()
for i in range(int(1e6)):
    act = agent.compute_action(obs)
    obs_next, rew, done, _ = env.step(act)
    buffer.store(obs, act, obs_next, rew, done)
    obs = obs_next
    if i % 1000 == 0:
        b_s, b_a, b_s_, b_r, b_d = buffer.get(size=64)
        # compute 2-step returns. How?
        b_ret = compute_2_step_return(buffer, b_r, b_d, ...)
        # update DQN policy
        agent.update(b_s, b_a, b_s_, b_r, b_d, b_ret)
```

Thus, we need a time-related interface for calculating the 2-step return. *process_fn()* finishes this work by providing the replay buffer, the sample index, and the sample batch data. Since we store all the data in the order of time, you can simply compute the 2-step return as:

```
class DQN_2step(BasePolicy):
    """some code"""

    def process_fn(self, batch, buffer, indices):
        buffer_len = len(buffer)
        batch_2 = buffer[(indices + 2) % buffer_len]
        # this will return a batch data where batch_2.obs is s_t+2
        # we can also get s_t+2 through:
        #   batch_2_obs = buffer.obs[(indices + 2) % buffer_len]
        # in short, buffer.obs[i] is equal to buffer[i].obs, but the former is more
↪effecient.
        Q = self(batch_2, eps=0)  # shape: [batchsize, action_shape]
        maxQ = Q.max(dim=-1)
        batch.returns = batch.rew \
            + self._gamma * buffer.rew[(indices + 1) % buffer_len] \
            + self._gamma ** 2 * maxQ
        return batch
```

This code does not consider the done flag, so it may not work very well. It shows two ways to get s_{t+2} from the replay buffer easily in *process_fn()*.

For other method, you can check out *tianshou.policy*. We give the usage of policy class a high-level explanation in *A High-level Explanation*.

1.3.4 Collector

The *Collector* enables the policy to interact with different types of environments conveniently.

collect() is the main method of Collector: it let the policy perform a specified number of step `n_step` or episode `n_episode` and store the data in the replay buffer, then return the statistics of the collected data such as episode's total reward.

The general explanation is listed in *A High-level Explanation*. Other usages of collector are listed in *Collector* documentation. Here are some example usages:

```
policy = PGPolicy(...)  # or other policies if you wish
env = gym.make("CartPole-v0")

replay_buffer = ReplayBuffer(size=10000)

# here we set up a collector with a single environment
collector = Collector(policy, env, buffer=replay_buffer)

# the collector supports vectorized environments as well
vec_buffer = VectorReplayBuffer(total_size=10000, buffer_num=3)
# buffer_num should be equal to (suggested) or larger than #envs
envs = DummyVectorEnv([lambda: gym.make("CartPole-v0") for _ in range(3)])
collector = Collector(policy, envs, buffer=vec_buffer)

# collect 3 episodes
collector.collect(n_episode=3)
# collect at least 2 steps
collector.collect(n_step=2)
# collect episodes with visual rendering ("render" is the sleep time between
# rendering consecutive frames)
collector.collect(n_episode=1, render=0.03)
```

There is also another type of collector *AsyncCollector* which supports asynchronous environment setting (for those taking a long time to step). However, AsyncCollector only supports **at least** `n_step` or `n_episode` collection due to the property of asynchronous environments.

1.3.5 Trainer

Once you have a collector and a policy, you can start writing the training method for your RL agent. Trainer, to be honest, is a simple wrapper. It helps you save energy for writing the training loop. You can also construct your own trainer: *Train a Policy with Customized Codes*.

Tianshou has three types of trainer: *onpolicy_trainer()* for on-policy algorithms such as Policy Gradient, *offpolicy_trainer()* for off-policy algorithms such as DQN, and *offline_trainer()* for offline algorithms such as BCQ. Please check out *tianshou.trainer* for the usage.

We also provide the corresponding iterator-based trainer classes *OnpolicyTrainer*, *OffpolicyTrainer*, *OfflineTrainer* to facilitate users writing more flexible training logic:

```
trainer = OnpolicyTrainer(...)
for epoch, epoch_stat, info in trainer:
    print(f"Epoch: {epoch}")
    print(epoch_stat)
    print(info)
```

(continues on next page)

(continued from previous page)

```
    do_something_with_policy()
    query_something_about_policy()
    make_a_plot_with(epoch_stat)
    display(info)

# or even iterate on several trainers at the same time

trainer1 = OnpolicyTrainer(...)
trainer2 = OnpolicyTrainer(...)
for result1, result2, ... in zip(trainer1, trainer2, ...):
    compare_results(result1, result2, ...)
```

1.3.6 A High-level Explanation

We give a high-level explanation through the pseudocode used in section *policy.process_fn*:

```
# pseudocode, cannot work                                   # methods in tianshou
obs = env.reset()
buffer = Buffer(size=10000)                                 # buffer = tianshou.data.
↪ReplayBuffer(size=10000)
agent = DQN()                                               # policy.__init__(...)
for i in range(int(1e6)):                                   # done in trainer
    act = agent.compute_action(obs)                         # act = policy(batch, ...
↪).act
    obs_next, rew, done, _ = env.step(act)                  # collector.collect(...)
    buffer.store(obs, act, obs_next, rew, done)             # collector.collect(...)
    obs = obs_next                                          # collector.collect(...)
    if i % 1000 == 0:                                       # done in trainer
                                                            # the following is done␣
↪in policy.update(batch_size, buffer)
        b_s, b_a, b_s_, b_r, b_d = buffer.get(size=64)      # batch, indices =␣
↪buffer.sample(batch_size)
        # compute 2-step returns. How?
        b_ret = compute_2_step_return(buffer, b_r, b_d, ...)    # policy.process_
↪fn(batch, buffer, indices)
        # update DQN policy
        agent.update(b_s, b_a, b_s_, b_r, b_d, b_ret)       # policy.learn(batch, ...
↪)
```

1.3.7 Conclusion

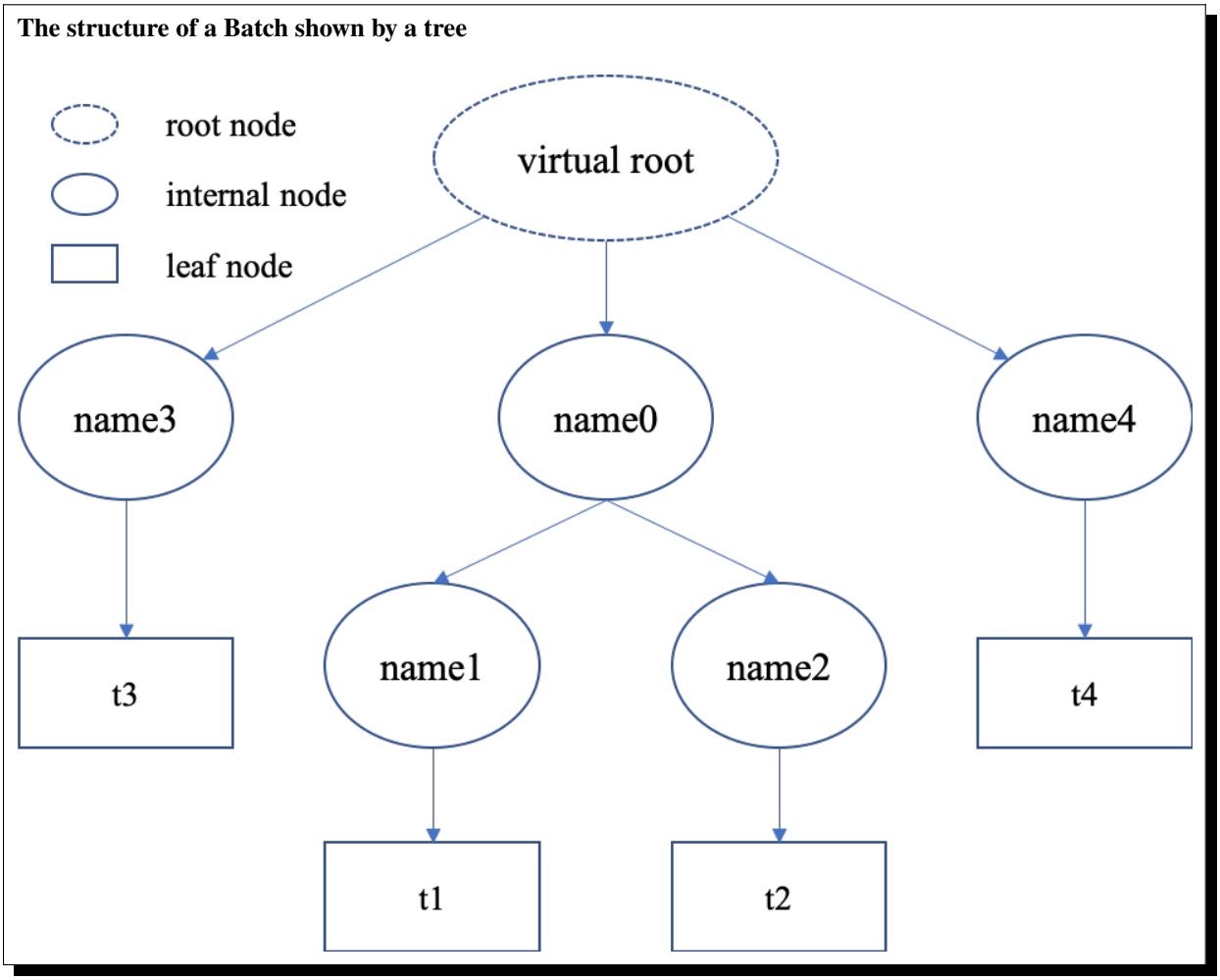
So far, we go through the overall framework of Tianshou. Really simple, isn't it?

1.4 Understand Batch

`Batch` is the internal data structure extensively used in Tianshou. It is designed to store and manipulate hierarchical named tensors. This tutorial aims to help users correctly understand the concept and the behavior of `Batch` so that users can make the best of Tianshou.

The tutorial has three parts. We first explain the concept of hierarchical named tensors, and introduce basic usage of `Batch`, followed by advanced topics of `Batch`.

1.4.1 Hierarchical Named Tensors



"Hierarchical named tensors" refers to a set of tensors where their names form a hierarchy. Suppose there are four tensors `[t1, t2, t3, t4]` with names `[name1, name2, name3, name4]`, where `name1` and `name2` belong to the same namespace `name0`, then the full name of tensor `t1` is `name0.name1`. That is, the hierarchy lies in the names of tensors.

We can describe the structure of hierarchical named tensors using a tree in the right. There is always a "virtual root" node to represent the whole object; internal nodes are keys (names), and leaf nodes are values (scalars or tensors).

Hierarchical named tensors are needed because we have to deal with the heterogeneity of reinforcement learning problems. The abstraction of RL is very simple, just:

```
state, reward, done = env.step(action)
```

`reward` and `done` are simple, they are mostly scalar values. However, the `state` and `action` vary with environments. For example, `state` can be simply a vector, a tensor, or a camera input combined with sensory input. In the last case, it is natural to store them as hierarchical named tensors. This hierarchy can go beyond `state` and `action`: we can store `state`, `action`, `reward`, and `done` together as hierarchical named tensors.

Note that, storing hierarchical named tensors is as easy as creating nested dictionary objects:

```
{
    'done': done,
    'reward': reward,
    'state': {
        'camera': camera,
        'sensory': sensory
    }
    'action': {
        'direct': direct,
        'point_3d': point_3d,
        'force': force,
    }
}
```

The real problem is how to **manipulate them**, such as adding new transition tuples into replay buffer and dealing with their heterogeneity. `Batch` is designed to easily create, store, and manipulate these hierarchical named tensors.

1.4.2 Basic Usages

Here we cover some basic usages of `Batch`, describing what `Batch` contains, how to construct `Batch` objects and how to manipulate them.

What Does Batch Contain

The content of `Batch` objects can be defined by the following rules.

1. A `Batch` object can be an empty `Batch()`, or have at least one key-value pairs. `Batch()` can be used to reserve keys, too. See *Key Reservations* for this advanced usage.
2. The keys are always strings (they are names of corresponding values).
3. The values can be scalars, tensors, or Batch objects. The recursive definition makes it possible to form a hierarchy of batches.
4. Tensors are the most important values. In short, tensors are n-dimensional arrays of the same data type. We support two types of tensors: PyTorch tensor type `torch.Tensor` and NumPy tensor type `np.ndarray`.
5. Scalars are also valid values. A scalar is a single boolean, number, or object. They can be python scalar (`False`, `1`, `2.3`, `None`, `'hello'`) or NumPy scalar (`np.bool_(True)`, `np.int32(1)`, `np.float64(2.3)`). They just shouldn't be mixed up with Batch/dict/tensors.

Note: `Batch` cannot store `dict` objects, because internally `Batch` uses `dict` to store data. During construction, `dict` objects will be automatically converted to `Batch` objects.

The data types of tensors are bool and numbers (any size of int and float as long as they are supported by NumPy or PyTorch). Besides, NumPy supports ndarray of objects and we take advantage of this feature to store non-number objects in `Batch`. If one wants to store data that are neither boolean nor numbers (such as strings and sets), they can store the data in `np.ndarray` with the `np.object` data type. This way, `Batch` can store any type of python objects.

Construction of Batch

There are two ways to construct a `Batch` object: from a `dict`, or using `kwargs`. Below are some code snippets.

```
>>> # directly passing a dict object (possibly nested) is ok
>>> data = Batch({'a': 4, 'b': [5, 5], 'c': '2312312'})
>>> # the list will automatically be converted to numpy array
>>> data.b
array([5, 5])
>>> data.b = np.array([3, 4, 5])
>>> print(data)
Batch(
    a: 4,
    b: array([3, 4, 5]),
    c: '2312312',
)
>>> # a list of dict objects (possibly nested) will be automatically stacked
>>> data = Batch([{'a': 0.0, 'b': "hello"}, {'a': 1.0, 'b': "world"}])
>>> print(data)
Batch(
    a: array([0., 1.]),
    b: array(['hello', 'world'], dtype=object),
)
```

```
>>> # construct a Batch with keyword arguments
>>> data = Batch(a=[4, 4], b=[5, 5], c=[None, None])
>>> print(data)
Batch(
    a: array([4, 4]),
    b: array([5, 5]),
    c: array([None, None], dtype=object),
)
>>> # combining keyword arguments and batch_dict works fine
>>> data = Batch({'a':[4, 4], 'b':[5, 5]}, c=[None, None]) # the first argument is a␣
↪dict, and 'c' is a keyword argument
>>> print(data)
Batch(
    a: array([4, 4]),
    b: array([5, 5]),
    c: array([None, None], dtype=object),
)
>>> arr = np.zeros((3, 4))
>>> # By default, Batch only keeps the reference to the data, but it also supports data␣
```

(continues on next page)

(continued from previous page)

```
↪copying
>>> data = Batch(arr=arr, copy=True) # data.arr now is a copy of 'arr'
```

Data Manipulation With Batch

Users can access the internal data by `b.key` or `b[key]`, where `b.key` finds the sub-tree with `key` as the root node. If the result is a sub-tree with non-empty keys, the key-reference can be chained, i.e. `b.key.key1.key2.key3`. When it reaches a leaf node, users get the data (scalars/tensors) stored in that `Batch` object.

```
>>> data = Batch(a=4, b=[5, 5])
>>> print(data.b)
[5 5]
>>> # obj.key is equivalent to obj["key"]
>>> print(data["a"])
4
>>> # iterating over data items like a dict is supported
>>> for key, value in data.items():
>>>     print(f"{key}: {value}")
a: 4
b: [5, 5]
>>> # obj.keys() and obj.values() work just like dict.keys() and dict.values()
>>> for key in data.keys():
>>>     print(f"{key}")
a
b
>>> # obj.update() behaves like dict.update()
>>> # this is the same as data.c = 1; data.c = 2; data.e = 3;
>>> data.update(c=1, d=2, e=3)
>>> print(data)
Batch(
    a: 4,
    b: array([5, 5]),
    c: 1,
    d: 2,
    e: 3,
)
```

Note: If `data` is a `dict` object, `for x in data` iterates over keys in the dict. However, it has a different meaning for `Batch` objects: `for x in data` iterates over `data[0], data[1], ..., data[-1]`. An example is given below.

`Batch` also partially reproduces the NumPy ndarray APIs. It supports advanced slicing, such as `batch[:, i]` so long as the slice is valid. Broadcast mechanism of NumPy works for `Batch`, too.

```
>>> # initialize Batch with tensors
>>> data = Batch(a=np.array([[0.0, 2.0], [1.0, 3.0]]), b=[[5, -5], [1, -2]])
>>> # if values have the same length/shape, that length/shape is used for this Batch
>>> # else, check the advanced topic for details
>>> print(len(data))
2
>>> print(data.shape)
```

(continues on next page)

(continued from previous page)

```
[2, 2]
>>> # access the first item of all the stored tensors, while keeping the structure of␣
↪Batch
>>> print(data[0])
Batch(
    a: array([0., 2.])
    b: array([ 5, -5]),
)
>>> # iterates over ``data[0], data[1], ..., data[-1]``
>>> for sample in data:
>>>     print(sample.a)
[0. 2.]
[1. 3.]

>>> # Advanced slicing works just fine
>>> # Arithmetic operations are passed to each value in the Batch, with broadcast enabled
>>> data[:, 1] += 1
>>> print(data)
Batch(
    a: array([[0., 3.],
              [1., 4.]]),
    b: array([[ 5, -4]]),
)

>>> # amazingly, you can directly apply np.mean to a Batch object
>>> print(np.mean(data))
Batch(
    a: 1.5,
    b: -0.25,
)

>>> # directly converted to a list is also available
>>> list(data)
[Batch(
     a: array([0., 3.]),
     b: array([ 5, -4]),
 ),
 Batch(
     a: array([1., 4.]),
     b: array([ 1, -1]),
 )]
```

Stacking and concatenating multiple `Batch` instances, or split an instance into multiple batches, they are all easy and intuitive in Tianshou. For now, we stick to the aggregation (stack/concatenate) of homogeneous (same structure) batches. Stack/Concatenation of heterogeneous batches are discussed in *Aggregation of Heterogeneous Batches*.

```
>>> data_1 = Batch(a=np.array([0.0, 2.0]), b=5)
>>> data_2 = Batch(a=np.array([1.0, 3.0]), b=-5)
>>> data = Batch.stack((data_1, data_2))
>>> print(data)
Batch(
    b: array([ 5, -5]),
```

(continues on next page)

(continued from previous page)

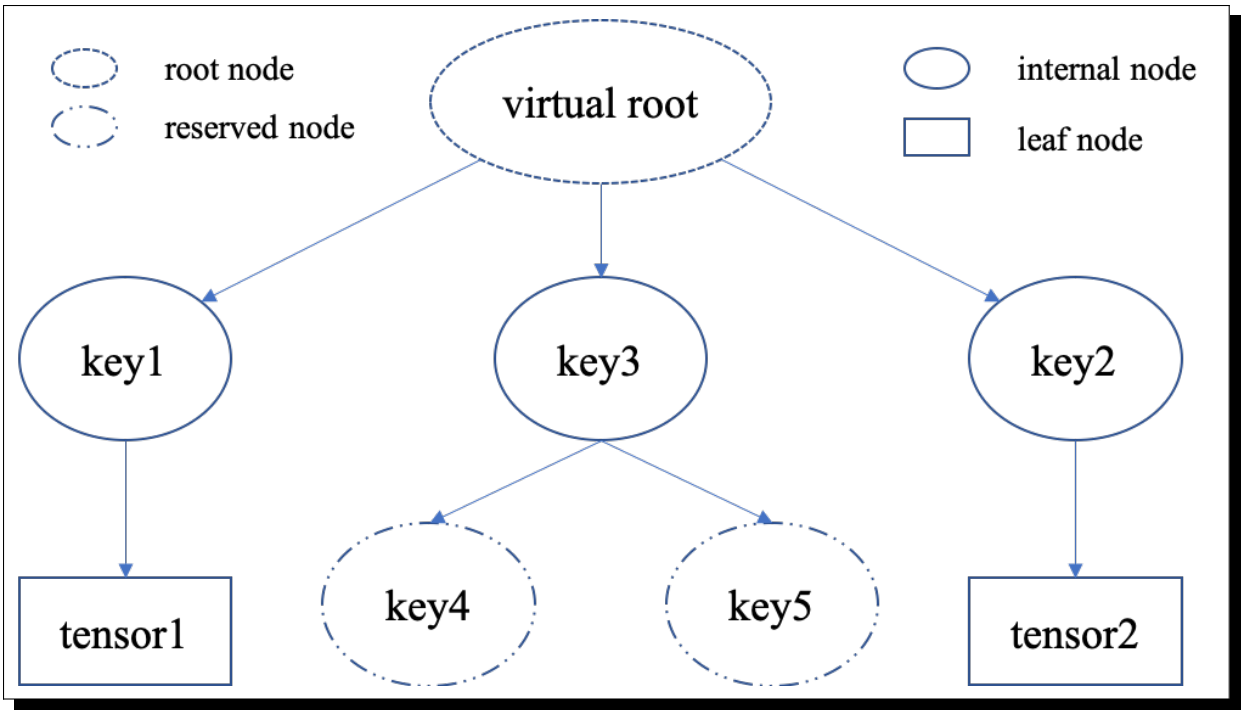
```
    a: array([[0., 2.],
              [1., 3.]]),
)
>>> # split supports random shuffling
>>> data_split = list(data.split(1, shuffle=False))
>>> print(list(data.split(1, shuffle=False)))
[Batch(
    b: array([5]),
    a: array([[0., 2.]]),
), Batch(
    b: array([-5]),
    a: array([[1., 3.]]),
)]
>>> data_cat = Batch.cat(data_split)
>>> print(data_cat)
Batch(
    b: array([ 5, -5]),
    a: array([[0., 2.],
              [1., 3.]]),
)
```

1.4.3 Advanced Topics

From here on, this tutorial focuses on advanced topics of `Batch`, including key reservation, length/shape, and aggregation of heterogeneous batches.

Key Reservations

The structure of a Batch with reserved keys



In many cases, we know in the first place what keys we have, but we do not know the shape of values until we run the environment. To deal with this, Tianshou supports key reservations: **reserve a key and use a placeholder value**.

The usage is easy: just use `Batch()` to be the value of reserved keys.

```
a = Batch(b=Batch()) # 'b' is a reserved key
# this is called hierarchical key reservation
a = Batch(b=Batch(c=Batch()), d=Batch()) # 'c' and 'd' are reserved key
# the structure of this last Batch is shown in the right figure
a = Batch(key1=tensor1, key2=tensor2, key3=Batch(key4=Batch(), key5=Batch()))
```

Still, we can use a tree (in the right) to show the structure of `Batch` objects with reserved keys, where reserved keys are special internal nodes that do not have attached leaf nodes.

Note: Reserved keys mean that in the future there will eventually be values attached to them. The values can be scalars, tensors, or even **Batch** objects. Understanding this is critical to understand the behavior of `Batch` when dealing with heterogeneous Batches.

The introduction of reserved keys gives rise to the need to check if a key is reserved. Tianshou provides `Batch.is_empty` to achieve this.

```
>>> Batch().is_empty()
True
>>> Batch(a=Batch(), b=Batch(c=Batch())).is_empty()
False
>>> Batch(a=Batch(), b=Batch(c=Batch())).is_empty(recurse=True)
True
>>> Batch(d=1).is_empty()
False
>>> Batch(a=np.float64(1.0)).is_empty()
False
```

The `Batch.is_empty` function has an option to decide whether to identify direct emptiness (just a `Batch()`) or to identify recursive emptiness (a `Batch` object without any scalar/tensor leaf nodes).

Note: Do not get confused with `Batch.is_empty` and `Batch.empty`. `Batch.empty` and its in-place variant `Batch.empty_` are used to set some values to zeros or None. Check the API documentation for further details.

Length and Shape

The most common usage of `Batch` is to store a Batch of data. The term "Batch" comes from the deep learning community to denote a mini-batch of sampled data from the whole dataset. In this regard, "Batch" typically means a collection of tensors whose first dimensions are the same. Then the length of a `Batch` object is simply the batch-size.

If all the leaf nodes in a `Batch` object are tensors, but they have different lengths, they can be readily stored in `Batch`. However, for `Batch` of this kind, the `len(obj)` seems a bit ambiguous. Currently, Tianshou returns the length of the shortest tensor, but we strongly recommend that users do not use the `len(obj)` operator on `Batch` objects with tensors of different lengths.

```
>>> data = Batch(a=[5., 4.], b=np.zeros((2, 3, 4)))
>>> data.shape
[2]
>>> len(data)
2
>>> data[0].shape
[]
>>> len(data[0])
TypeError: Object of type 'Batch' has no len()
```

Note: Following the convention of scientific computation, scalars have no length. If there is any scalar leaf node in a `Batch` object, an exception will occur when users call `len(obj)`.

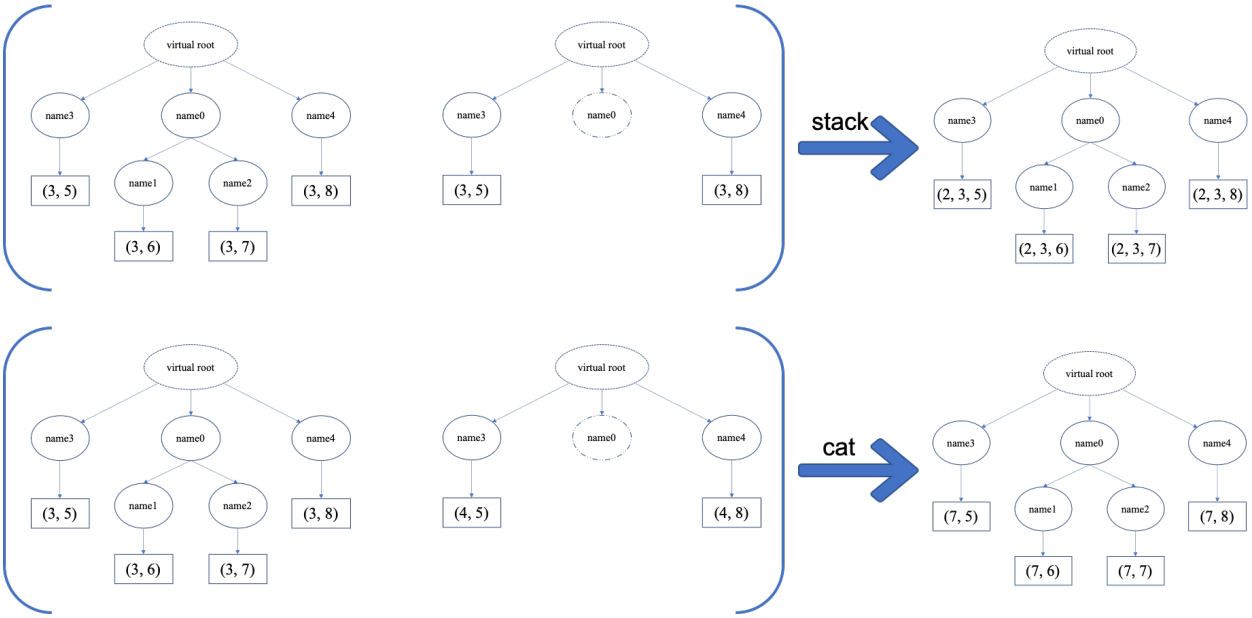
Besides, values of reserved keys are undetermined, so they have no length, neither. Or, to be specific, values of reserved keys have lengths of **any**. When there is a mix of tensors and reserved keys, the latter will be ignored in `len(obj)` and the minimum length of tensors is returned. When there is not any tensor in the `Batch` object, Tianshou raises an exception, too.

The `obj.shape` attribute of `Batch` behaves somewhat similar to `len(obj)`:

1. If all the leaf nodes in a `Batch` object are tensors with the same shape, that shape is returned.
2. If all the leaf nodes in a `Batch` object are tensors but they have different shapes, the minimum length of each dimension is returned.
3. If there is any scalar value in a `Batch` object, `obj.shape` returns `[]`.
4. The shape of reserved keys is undetermined, too. We treat their shape as `[]`.

Aggregation of Heterogeneous Batches

In this section, we talk about aggregation operators (stack/concatenate) on heterogeneous `Batch` objects. The following picture will give you an intuitive understanding of this behavior. It shows two examples of aggregation operators with heterogeneous `Batch`. The shapes of tensors are annotated in the leaf nodes.



We only consider the heterogeneity in the structure of `Batch` objects. The aggregation operators are eventually done by NumPy/PyTorch operators (`np.stack`, `np.concatenate`, `torch.stack`, `torch.cat`). Heterogeneity in values can fail these operators (such as stacking `np.ndarray` with `torch.Tensor`, or stacking tensors with different shapes) and an exception will be raised.

The behavior is natural: for keys that are not shared across all batches, batches that do not have these keys will be padded by zeros (or `None` if the data type is `np.object`). It can be written in the following scripts:

```
>>> # examples of stack: a is missing key `b`, and b is missing key `a`
>>> a = Batch(a=np.zeros([4, 4]), common=Batch(c=np.zeros([4, 5])))
>>> b = Batch(b=np.zeros([4, 6]), common=Batch(c=np.zeros([4, 5])))
>>> c = Batch.stack([a, b])
>>> c.a.shape
(2, 4, 4)
>>> c.b.shape
(2, 4, 6)
>>> c.common.c.shape
(2, 4, 5)
>>> # None or 0 is padded with appropriate shape
>>> data_1 = Batch(a=np.array([0.0, 2.0]))
>>> data_2 = Batch(a=np.array([1.0, 3.0]), b='done')
>>> data = Batch.stack((data_1, data_2))
>>> print(data)
Batch(
    a: array([[0., 2.],
              [1., 3.]]),
    b: array([None, 'done'], dtype=object),
)
```

(continues on next page)

(continued from previous page)

```
>>> # examples of cat: a is missing key `b`, and b is missing key `a`
>>> a = Batch(a=np.zeros([3, 4]), common=Batch(c=np.zeros([3, 5])))
>>> b = Batch(b=np.zeros([4, 3]), common=Batch(c=np.zeros([4, 5])))
>>> c = Batch.cat([a, b])
>>> c.a.shape
(7, 4)
>>> c.b.shape
(7, 3)
>>> c.common.c.shape
(7, 5)
```

However, there are some cases when batches are too heterogeneous that they cannot be aggregated:

```
>>> a = Batch(a=np.zeros([4, 4]))
>>> b = Batch(a=Batch(b=Batch()))
>>> # this will raise an exception
>>> c = Batch.stack([a, b])
```

Then how to determine if batches can be aggregated? Let's rethink the purpose of reserved keys. What is the advantage of `a1=Batch(b=Batch())` over `a2=Batch()`? The only difference is that `a1.b` returns `Batch()` but `a2.b` raises an exception. That's to say, **we reserve keys for attribute reference**.

We say a key chain `k=[key1, key2, ..., keyn]` applies to `b` if the expression `b.key1.key2.{...}.keyn` is valid, and the result is `b[k]`.

For a set of `Batch` objects denoted as S, they can be aggregated if there exists a `Batch` object `b` satisfying the following rules:

1. Key chain applicability: For any object `bi` in S, and any key chain `k`, if `bi[k]` is valid, then `b[k]` is valid.
2. Type consistency: If `bi[k]` is not `Batch()` (the last key in the key chain is not a reserved key), then the type of `b[k]` should be the same as `bi[k]` (both should be scalar/tensor/non-empty Batch values).

The `Batch` object `b` satisfying these rules with the minimum number of keys determines the structure of aggregating S. The values are relatively easy to define: for any key chain `k` that applies to `b`, `b[k]` is the stack/concatenation of `[bi[k] for bi in S]` (if `k` does not apply to `bi`, the appropriate size of zeros or `None` are filled automatically). If `bi[k]` are all `Batch()`, then the aggregation result is also an empty `Batch()`.

Miscellaneous Notes

1. `Batch` is serializable and therefore Pickle compatible. `Batch` objects can be saved to disk and later restored by the python `pickle` module. This pickle compatibility is especially important for distributed sampling from environments.

```
>>> data = Batch(a=np.zeros((3, 4)))
>>> data.to_torch(dtype=torch.float32, device='cpu')
>>> print(data.a)
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
>>> # data.to_numpy is also available
>>> data.to_numpy()
```

2. It is often the case that the observations returned from the environment are all NumPy ndarray but the policy requires `torch.Tensor` for prediction and learning. In this regard, Tianshou provides helper functions to convert

the stored data in-place into Numpy arrays or Torch tensors.

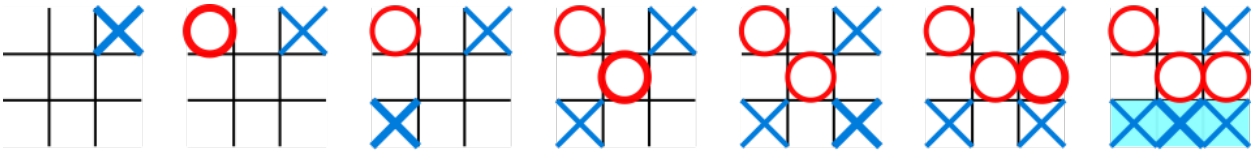
3. `obj.stack_([a, b])` is the same as `Batch.stack([obj, a, b])`, and `obj.cat_([a, b])` is the same as `Batch.cat([obj, a, b])`. Considering the frequent requirement of concatenating two `Batch` objects, Tianshou also supports `obj.cat_(a)` to be an alias of `obj.cat_([a])`.
4. `Batch.cat` and `Batch.cat_` does not support `axis` argument as `np.concatenate` and `torch.cat` currently.
5. `Batch.stack` and `Batch.stack_` support the `axis` argument so that one can stack batches besides the first dimension. But be cautious, if there are keys that are not shared across all batches, `stack` with `axis != 0` is undefined, and will cause an exception currently.

1.5 Multi-Agent RL

Tianshou use *PettingZoo* environment for multi-agent RL training. Here are some helpful tutorial links:

- https://pettingzoo.farama.org/tutorials/tianshou/beginner/
- https://pettingzoo.farama.org/tutorials/tianshou/intermediate/
- https://pettingzoo.farama.org/tutorials/tianshou/advanced/

In this section, we describe how to use Tianshou to implement multi-agent reinforcement learning. Specifically, we will design an algorithm to learn how to play Tic Tac Toe (see the image below) against a random opponent.



1.5.1 Tic-Tac-Toe Environment

The scripts are located at `test/pettingzoo/`. We have implemented *PettingZooEnv* which can wrap any PettingZoo environment. PettingZoo offers a 3x3 Tic-Tac-Toe environment, let's first explore it.

```
>>> from tianshou.env import PettingZooEnv       # wrapper for PettingZoo environments
>>> from pettingzoo.classic import tictactoe_v3  # the Tic-Tac-Toe environment to be␣
↪wrapped
>>> # This board has 3 rows and 3 cols (9 places in total)
>>> # Players place 'x' and 'o' in turn on the board
>>> # The player who first gets 3 consecutive 'x's or 'o's wins
>>>
>>> env = PettingZooEnv(tictactoe_v3.env(render_mode="human"))
>>> obs = env.reset()
>>> env.render()                                 # render the empty board
board (step 0):
     |     |
  -  |  -  |  -
_____|_____|_____
     |     |
  -  |  -  |  -
_____|_____|_____
     |     |
  -  |  -  |  -
```

(continues on next page)

(continued from previous page)

```
     |     |
>>> print(obs)                                    # let's see the shape of the observation
{'agent_id': 'player_1', 'obs': array([[[0, 0],
        [0, 0],
        [0, 0]],

       [[0, 0],
        [0, 0],
        [0, 0]],

       [[0, 0],
        [0, 0],
        [0, 0]]], dtype=int8), 'mask': [True, True, True, True, True, True, True, True,␣
↪True]}
```

The observation variable `obs` returned from the environment is a `dict`, with three keys `agent_id`, `obs`, `mask`. This is a general structure in multi-agent RL where agents take turns. The meaning of these keys are:

- `agent_id`: the id of the current acting agent. In our Tic-Tac-Toe case, the agent_id can be `player_1` or `player_2`.
- `obs`: the actual observation of the environment. In the Tic-Tac-Toe game above, the observation variable `obs` is a `np.ndarray` with the shape of (3, 3, 2). For `player_1`, the first 3x3 plane represents the placement of Xs, and the second plane shows the placement of Os. The possible values for each cell are 0 or 1; in the first plane, 1 indicates that an X has been placed in that cell, and 0 indicates that X is not in that cell. Similarly, in the second plane, 1 indicates that an O has been placed in that cell, while 0 indicates that an O has not been placed. For `player_2`, the observation is the same, but Xs and Os swap positions, so Os are encoded in plane 1 and Xs in plane 2.
- `mask`: the action mask in the current timestep. In board games or card games, the legal action set varies with time. The mask is a boolean array. For Tic-Tac-Toe, index `i` means the place of `i/N` th row and `i%N` th column. If `mask[i] == True`, the player can place an `x` or `o` at that position. Now the board is empty, so the mask is all the true, contains all the positions on the board.

Note: There is no special formulation of `mask` either in discrete action space or in continuous action space. You can also use some action spaces like `gymnasium.spaces.Discrete` or `gymnasium.spaces.Box` to represent the available action space. Currently, we use a boolean array.

Let's play two steps to have an intuitive understanding of the environment.

```
>>> import numpy as np
>>> action = 0                                    # action is either an integer, or an np.
↪ndarray with one element
>>> obs, reward, done, info = env.step(action)  # the env.step follows the api of␣
↪Gymnasium
>>> print(obs)                                    # notice the change in the observation
{'agent_id': 'player_2', 'obs': array([[[0, 1],
        [0, 0],
        [0, 0]],

       [[0, 0],
        [0, 0],
        [0, 0]],
```

(continues on next page)

(continued from previous page)

```
       [[0, 0],
        [0, 0],
        [0, 0]]], dtype=int8), 'mask': [False, True, True, True, True, True, True, True␣
↪True]}
>>> # reward has two items, one for each player: 1 for win, -1 for lose, and 0 otherwise
>>> print(reward)
[0. 0.]
>>> print(done)                                  # done indicates whether the game is over
False
>>> # info is always an empty dict in Tic-Tac-Toe, but may contain some useful␣
↪information in environments other than Tic-Tac-Toe.
>>> print(info)
{}
```

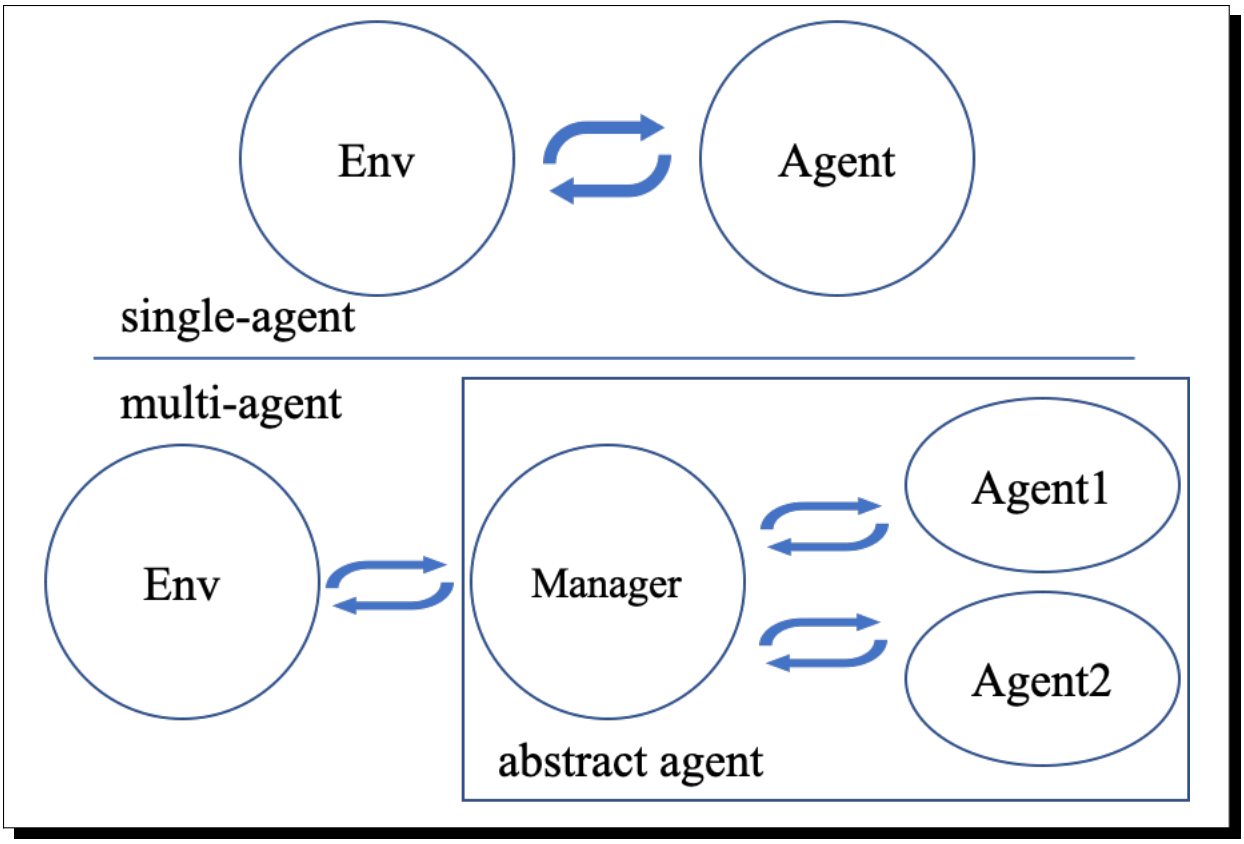
One worth-noting case is that the game is over when there is only one empty position, rather than when there is no position. This is because the player just has one choice (literally no choice) in this game.

```
>>> # omitted actions: 3, 1, 4
>>> obs, reward, done, info = env.step(2)  # player_1 wins
>>> print((reward, done))
([1, -1], True)
>>> env.render()
     |     |
  X  |  O  |  -
_____|_____|_____
     |     |
  X  |  O  |  -
_____|_____|_____
     |     |
  X  |  -  |  -
     |     |
```

After being familiar with the environment, let's try to play with random agents first!

1.5.2 Two Random Agents

The relationship between MultiAgentPolicyManager (Manager) and BasePolicy (Agent)



Tianshou already provides some builtin classes for multi-agent learning. You can check out the API documentation for details. Here we use *RandomPolicy* and *MultiAgentPolicyManager*. The figure on the right gives an intuitive explanation.

```
>>> from tianshou.data import Collector
>>> from tianshou.env import DummyVectorEnv
>>> from tianshou.policy import RandomPolicy, MultiAgentPolicyManager
>>>
>>> # agents should be wrapped into one policy,
>>> # which is responsible for calling the acting agent correctly
>>> # here we use two random agents
>>> policy = MultiAgentPolicyManager([RandomPolicy(), RandomPolicy()], env)
>>>
>>> # need to vectorize the environment for the collector
>>> env = DummyVectorEnv([lambda: env])
>>>
>>> # use collectors to collect a episode of trajectories
>>> # the reward is a vector, so we need a scalar metric to monitor the training
>>> collector = Collector(policy, env)
>>>
>>> # you will see a long trajectory showing the board status at each timestep
>>> result = collector.collect(n_episode=1, render=.1)
(only show the last 3 steps)
     |     |
  X  |  X  |  -
_____|_____|_____
```

(continues on next page)

(continued from previous page)

```
     |     |
  X  |  O  |  -
_____|_____|_____
     |     |
  O  |  -  |  -
     |     |
     |     |
  X  |  X  |  -
_____|_____|_____
     |     |
  X  |  O  |  -
_____|_____|_____
     |     |
  O  |  -  |  O
     |     |
     |     |
  X  |  X  |  X
_____|_____|_____
     |     |
  X  |  O  |  -
_____|_____|_____
     |     |
  O  |  -  |  O
     |     |
```

Random agents perform badly. In the above game, although agent 2 wins finally, it is clear that a smart agent 1 would place an `x` at row 4 col 4 to win directly.

1.5.3 Train an MARL Agent

So let's start to train our Tic-Tac-Toe agent! First, import some required modules.

```
import argparse
import os
from copy import deepcopy
from typing import Optional, Tuple

import gymnasium as gym
import numpy as np
import torch
from pettingzoo.classic import tictactoe_v3
from torch.utils.tensorboard import SummaryWriter

from tianshou.data import Collector, VectorReplayBuffer
from tianshou.env import DummyVectorEnv
from tianshou.env.pettingzoo_env import PettingZooEnv
from tianshou.policy import (
    BasePolicy,
    DQNPolicy,
    MultiAgentPolicyManager,
    RandomPolicy,
```

(continues on next page)

(continued from previous page)

```
)
from tianshou.trainer import offpolicy_trainer
from tianshou.utils import TensorboardLogger
from tianshou.utils.net.common import Net
```

The explanation of each Tianshou class/function will be deferred to their first usages. Here we define some arguments and hyperparameters of the experiment. The meaning of arguments is clear by just looking at their names.

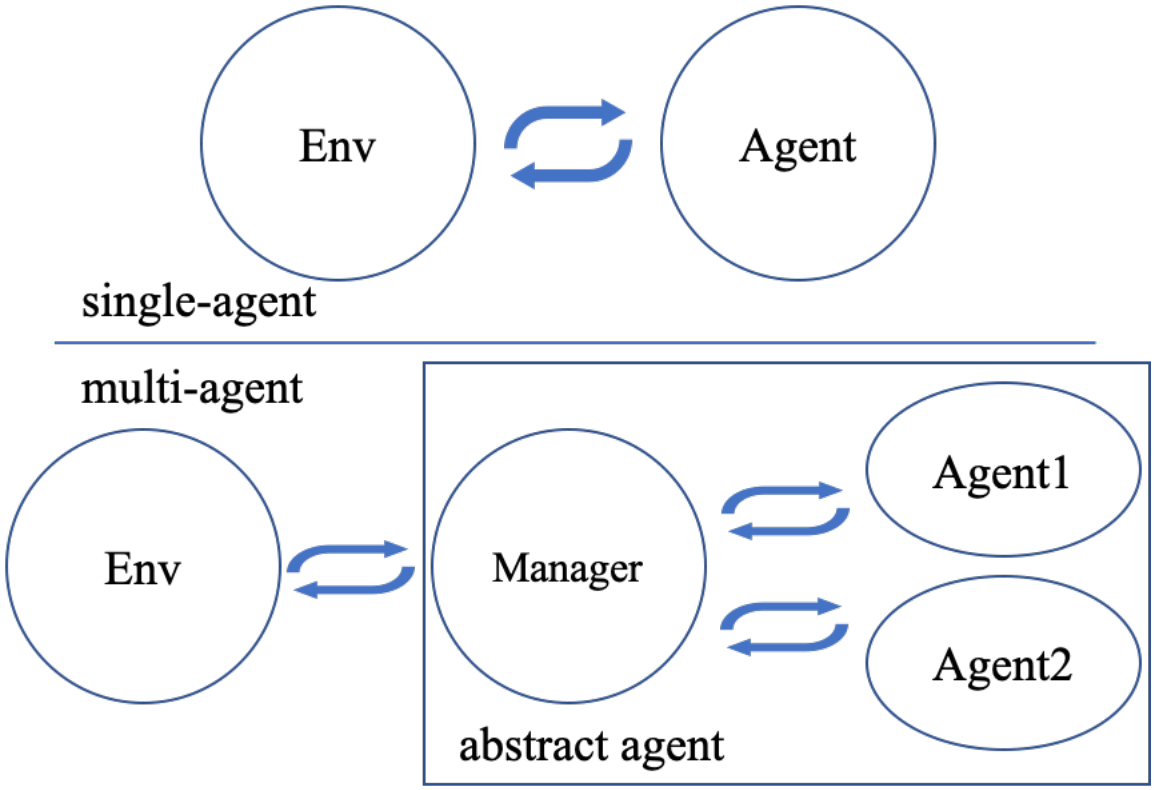
```
def get_parser() -> argparse.ArgumentParser:
    parser = argparse.ArgumentParser()
    parser.add_argument('--seed', type=int, default=1626)
    parser.add_argument('--eps-test', type=float, default=0.05)
    parser.add_argument('--eps-train', type=float, default=0.1)
    parser.add_argument('--buffer-size', type=int, default=20000)
    parser.add_argument('--lr', type=float, default=1e-4)
    parser.add_argument(
        '--gamma', type=float, default=0.9, help='a smaller gamma favors earlier win'
    )
    parser.add_argument('--n-step', type=int, default=3)
    parser.add_argument('--target-update-freq', type=int, default=320)
    parser.add_argument('--epoch', type=int, default=50)
    parser.add_argument('--step-per-epoch', type=int, default=1000)
    parser.add_argument('--step-per-collect', type=int, default=10)
    parser.add_argument('--update-per-step', type=float, default=0.1)
    parser.add_argument('--batch-size', type=int, default=64)
    parser.add_argument(
        '--hidden-sizes', type=int, nargs='*', default=[128, 128, 128, 128]
    )
    parser.add_argument('--training-num', type=int, default=10)
    parser.add_argument('--test-num', type=int, default=10)
    parser.add_argument('--logdir', type=str, default='log')
    parser.add_argument('--render', type=float, default=0.1)
    parser.add_argument(
        '--win-rate',
        type=float,
        default=0.6,
        help='the expected winning rate: Optimal policy can get 0.7'
    )
    parser.add_argument(
        '--watch',
        default=False,
        action='store_true',
        help='no training, '
        'watch the play of pre-trained models'
    )
    parser.add_argument(
        '--agent-id',
        type=int,
        default=2,
        help='the learned agent plays as the'
        ' agent_id-th player. Choices are 1 and 2.'
    )
```

(continues on next page)

(continued from previous page)

```
    parser.add_argument(
        '--resume-path',
        type=str,
        default='',
        help='the path of agent pth file '
        'for resuming from a pre-trained agent'
    )
    parser.add_argument(
        '--opponent-path',
        type=str,
        default='',
        help='the path of opponent agent pth file '
        'for resuming from a pre-trained agent'
    )
    parser.add_argument(
        '--device', type=str, default='cuda' if torch.cuda.is_available() else 'cpu'
    )
    return parser

def get_args() -> argparse.Namespace:
    parser = get_parser()
    return parser.parse_known_args()[0]
```

The relationship between MultiAgentPolicyManager (Manager) and BasePolicy (Agent)



The following `get_agents` function returns agents and their optimizers from either constructing a new policy, or loading from disk, or using the pass-in arguments. For the models:

- The action model we use is an instance of *Net*, essentially a multi-layer perceptron with the ReLU activation function;
- The network model is passed to a *DQNPolicy*, where actions are selected according to both the action mask and their Q-values;
- The opponent can be either a random agent *RandomPolicy* that randomly chooses an action from legal actions, or it can be a pre-trained *DQNPolicy* allowing learned agents to play with themselves.

Both agents are passed to *MultiAgentPolicyManager*, which is responsible to call the correct agent according to the `agent_id` in the observation. *MultiAgentPolicyManager* also dispatches data to each agent according to `agent_id`, so that each agent seems to play with a virtual single-agent environment.

Here it is:

```
def get_agents(
    args: argparse.Namespace = get_args(),
    agent_learn: Optional[BasePolicy] = None,
    agent_opponent: Optional[BasePolicy] = None,
    optim: Optional[torch.optim.Optimizer] = None,
) -> Tuple[BasePolicy, torch.optim.Optimizer, list]:
    env = get_env()
    observation_space = env.observation_space['observation'] if isinstance(
        env.observation_space, gym.spaces.Dict
    ) else env.observation_space
    args.state_shape = observation_space.shape or observation_space.n
    args.action_shape = env.action_space.shape or env.action_space.n
    if agent_learn is None:
        # model
        net = Net(
            args.state_shape,
            args.action_shape,
            hidden_sizes=args.hidden_sizes,
            device=args.device
        ).to(args.device)
        if optim is None:
            optim = torch.optim.Adam(net.parameters(), lr=args.lr)
        agent_learn = DQNPolicy(
            net,
            optim,
            args.gamma,
            args.n_step,
            target_update_freq=args.target_update_freq
        )
        if args.resume_path:
            agent_learn.load_state_dict(torch.load(args.resume_path))

    if agent_opponent is None:
        if args.opponent_path:
            agent_opponent = deepcopy(agent_learn)
            agent_opponent.load_state_dict(torch.load(args.opponent_path))
        else:
            agent_opponent = RandomPolicy()
```

(continues on next page)

(continued from previous page)

```
    if args.agent_id == 1:
        agents = [agent_learn, agent_opponent]
    else:
        agents = [agent_opponent, agent_learn]
    policy = MultiAgentPolicyManager(agents, env)
    return policy, optim, env.agents
```

With the above preparation, we are close to the first learned agent. The following code is almost the same as the code in the DQN tutorial.

```
def get_env(render_mode=None):
    return PettingZooEnv(tictactoe_v3.env(render_mode=render_mode))


def train_agent(
    args: argparse.Namespace = get_args(),
    agent_learn: Optional[BasePolicy] = None,
    agent_opponent: Optional[BasePolicy] = None,
    optim: Optional[torch.optim.Optimizer] = None,
) -> Tuple[dict, BasePolicy]:

    # ======== environment setup =========
    train_envs = DummyVectorEnv([get_env for _ in range(args.training_num)])
    test_envs = DummyVectorEnv([get_env for _ in range(args.test_num)])
    # seed
    np.random.seed(args.seed)
    torch.manual_seed(args.seed)
    train_envs.seed(args.seed)
    test_envs.seed(args.seed)

    # ======== agent setup =========
    policy, optim, agents = get_agents(
        args, agent_learn=agent_learn, agent_opponent=agent_opponent, optim=optim
    )

    # ======== collector setup =========
    train_collector = Collector(
        policy,
        train_envs,
        VectorReplayBuffer(args.buffer_size, len(train_envs)),
        exploration_noise=True
    )
    test_collector = Collector(policy, test_envs, exploration_noise=True)
    # policy.set_eps(1)
    train_collector.collect(n_step=args.batch_size * args.training_num)

    # ======== tensorboard logging setup =========
    log_path = os.path.join(args.logdir, 'tic_tac_toe', 'dqn')
    writer = SummaryWriter(log_path)
    writer.add_text("args", str(args))
    logger = TensorboardLogger(writer)
```

(continues on next page)

(continued from previous page)

```
    # ======== callback functions used during training =========
    def save_best_fn(policy):
        if hasattr(args, 'model_save_path'):
            model_save_path = args.model_save_path
        else:
            model_save_path = os.path.join(
                args.logdir, 'tic_tac_toe', 'dqn', 'policy.pth'
            )
        torch.save(
            policy.policies[agents[args.agent_id - 1]].state_dict(), model_save_path
        )

    def stop_fn(mean_rewards):
        return mean_rewards >= args.win_rate

    def train_fn(epoch, env_step):
        policy.policies[agents[args.agent_id - 1]].set_eps(args.eps_train)

    def test_fn(epoch, env_step):
        policy.policies[agents[args.agent_id - 1]].set_eps(args.eps_test)

    def reward_metric(rews):
        return rews[:, args.agent_id - 1]

    # trainer
    result = offpolicy_trainer(
        policy,
        train_collector,
        test_collector,
        args.epoch,
        args.step_per_epoch,
        args.step_per_collect,
        args.test_num,
        args.batch_size,
        train_fn=train_fn,
        test_fn=test_fn,
        stop_fn=stop_fn,
        save_best_fn=save_best_fn,
        update_per_step=args.update_per_step,
        logger=logger,
        test_in_train=False,
        reward_metric=reward_metric
    )

    return result, policy.policies[agents[args.agent_id - 1]]

# ======== a test function that tests a pre-trained agent ======
def watch(
    args: argparse.Namespace = get_args(),
    agent_learn: Optional[BasePolicy] = None,
    agent_opponent: Optional[BasePolicy] = None,
```

(continues on next page)

(continued from previous page)

```
) -> None:
    env = get_env(render_mode="human")
    env = DummyVectorEnv([lambda: env])
    policy, optim, agents = get_agents(
        args, agent_learn=agent_learn, agent_opponent=agent_opponent
    )
    policy.eval()
    policy.policies[agents[args.agent_id - 1]].set_eps(args.eps_test)
    collector = Collector(policy, env, exploration_noise=True)
    result = collector.collect(n_episode=1, render=args.render)
    rews, lens = result["rews"], result["lens"]
    print(f"Final reward: {rews[:, args.agent_id - 1].mean()}, length: {lens.mean()}")

# train the agent and watch its performance in a match!
args = get_args()
result, agent = train_agent(args)
watch(args, agent)
```

That's it. By executing the code, you will see a progress bar indicating the progress of training. After about less than 1 minute, the agent has finished training, and you can see how it plays against the random agent. Here is an example:

```
     |     |
  -  |  -  |  -
_____|_____|_____
     |     |
  -  |  -  |  X
_____|_____|_____
     |     |
  -  |  -  |  -
     |     |
     |     |
  -  |  -  |  -
_____|_____|_____
     |     |
  -  |  O  |  X
_____|_____|_____
     |     |
  -  |  -  |  -
     |     |
     |     |
  -  |  -  |  -
_____|_____|_____
     |     |
  X  |  O  |  X
_____|_____|_____
     |     |
  -  |  -  |  -
     |     |
     |     |
  -  |  O  |  -
_____|_____|_____
     |     |
```

(continues on next page)

(continued from previous page)

```
  X  |  O  |  X
_____|_____|_____
     |     |
  -  |  -  |  -
     |     |
     |     |
  -  |  O  |  -
_____|_____|_____
     |     |
  X  |  O  |  X
_____|_____|_____
     |     |
  -  |  X  |  -
     |     |
     |     |
  O  |  O  |  -
_____|_____|_____
     |     |
  X  |  O  |  X
_____|_____|_____
     |     |
  -  |  X  |  -
     |     |
     |     |
  O  |  O  |  X
_____|_____|_____
     |     |
  X  |  O  |  X
_____|_____|_____
     |     |
  -  |  X  |  -
     |     |
     |     |
  O  |  O  |  X
_____|_____|_____
     |     |
  X  |  O  |  X
_____|_____|_____
     |     |
  -  |  X  |  O
     |     |
Final reward: 1.0, length: 8.0
```

Notice that, our learned agent plays the role of agent 2, placing `o` on the board. The agent performs pretty well against the random opponent! It learns the rule of the game by trial and error, and learns that four consecutive `o` means winning, so it does!

The above code can be executed in a python shell or can be saved as a script file (we have saved it in `test/pettingzoo/test_tic_tac_toe.py`). In the latter case, you can train an agent by

```
$ python test_tic_tac_toe.py
```

By default, the trained agent is stored in `log/tic_tac_toe/dqn/policy.pth`. You can also make the trained agent

play against itself, by

```
$ python test_tic_tac_toe.py --watch --resume-path log/tic_tac_toe/dqn/policy.pth --
↪opponent-path log/tic_tac_toe/dqn/policy.pth
```

Here is our output:

```
      |       |
  -   |   -   |   -
______|_______|______
      |       |
  -   |   X   |   -
______|_______|______
      |       |
  -   |   -   |   -
      |       |
      |       |
  -   |   O   |   -
______|_______|______
      |       |
  -   |   X   |   -
______|_______|______
      |       |
  -   |   -   |   -
      |       |
      |       |
  X   |   O   |   -
______|_______|______
      |       |
  -   |   X   |   -
______|_______|______
      |       |
  -   |   -   |   -
      |       |
      |       |
  X   |   O   |   -
______|_______|______
      |       |
  -   |   X   |   -
______|_______|______
      |       |
  -   |   -   |   O
      |       |
      |       |
  X   |   O   |   -
______|_______|______
      |       |
  -   |   X   |   -
______|_______|______
      |       |
  -   |   X   |   O
      |       |
      |       |
  X   |   O   |   O
```

(continues on next page)

(continued from previous page)

```
_____|_____|_____
     |     |
  -  |  X  |  -
_____|_____|_____
     |     |
  -  |  X  |  O
     |     |
     |     |
  X  |  O  |  O
_____|_____|_____
     |     |
  -  |  X  |  -
_____|_____|_____
     |     |
  X  |  X  |  O
     |     |
     |     |
  X  |  O  |  O
_____|_____|_____
     |     |
  -  |  X  |  O
_____|_____|_____
     |     |
  X  |  X  |  O
     |     |
Final reward: 1.0, length: 8.0
```

Well, although the learned agent plays well against the random agent, it is far away from intelligence.

Next, maybe you can try to build more intelligent agents by letting the agent learn from self-play, just like AlphaZero!

In this tutorial, we show an example of how to use Tianshou for multi-agent RL. Tianshou is a flexible and easy to use RL library. Make the best of Tianshou by yourself!

1.6 Logging Experiments

Tianshou comes with multiple experiment tracking and logging solutions to manage and reproduce your experiments. The dashboard loggers currently available are:

- *TensorboardLogger*
- *WandbLogger*
- *LazyLogger*

1.6.1 TensorboardLogger

Tensorboard tracks your experiment metrics in a local dashboard. Here is how you can use TensorboardLogger in your experiment:

```
from torch.utils.tensorboard import SummaryWriter
from tianshou.utils import TensorboardLogger

log_path = os.path.join(args.logdir, args.task, "dqn")
writer = SummaryWriter(log_path)
writer.add_text("args", str(args))
logger = TensorboardLogger(writer)
result = trainer(..., logger=logger)
```

1.6.2 WandbLogger

WandbLogger can be used to visualize your experiments in a hosted W&B dashboard. It can be installed via `pip install wandb`. You can also save your checkpoints in the cloud and restore your runs from those checkpoints. Here is how you can enable WandbLogger:

```
from tianshou.utils import WandbLogger
from torch.utils.tensorboard import SummaryWriter

logger = WandbLogger(...)
writer = SummaryWriter(log_path)
writer.add_text("args", str(args))
logger.load(writer)
result = trainer(..., logger=logger)
```

Please refer to *WandbLogger* documentation for advanced configuration.

For logging checkpoints on any device, you need to define a `save_checkpoint_fn` which saves the experiment checkpoint and returns the path of the saved checkpoint:

```
def save_checkpoint_fn(epoch, env_step, gradient_step):
    ckpt_path = ...
    # save model
    return ckpt_path
```

Then, use this function with `WandbLogger` to automatically version your experiment checkpoints after every `save_interval` step.

For resuming runs from checkpoint artifacts on any device, pass the W&B `run_id` of the run that you want to continue in `WandbLogger`. It will then download the latest version of the checkpoint and resume your runs from the checkpoint.

The example scripts are under test_psrl.py and atari_dqn.py.

1.6.3 LazyLogger

This is a place-holder logger that does nothing.

1.7 Benchmark

1.7.1 Mujoco Benchmark

Tianshou's Mujoco benchmark contains state-of-the-art results.

Every experiment is conducted under 10 random seeds for 1-10M steps. Please refer to https://github.com/thu-ml/tianshou/tree/master/examples/mujoco for source code and detailed results.

The table below compares the performance of Tianshou against published results on OpenAI Gym MuJoCo benchmarks. We use max average return in 1M timesteps as the reward metric. ~ means the result is approximated from the plots because quantitative results are not provided. - means results are not provided. The best-performing baseline on each task is highlighted in boldface. Referenced baselines include TD3 paper, SAC paper, PPO paper, ACKTR paper, OpenAI Baselines and Spinning Up.

Task		Ant	HalfCheetah	Hopper	Walker2d	Swimmer	Humanoid	Reacher	IPendulum	IDPendulum
DDPG	Tianshou	990.4	**11718.7**	**2197.0**	1400.6	**144.1**	**177.3**	**-3.3**	**1000.0**	8364.3
	TD3 Paper	**1005.3**	3305.6	**2020.5**	1843.6	/	/	-6.5	**1000.0**	**9355.5**
	TD3 Paper (Our)	888.8	8577.3	1860.0	**3098.1**	/	/	-4.0	**1000.0**	8370.0
	Spinning Up	~840	~11000	~1800	~1950	~137	/	/	/	/
TD3	Tianshou	**5116.4**	**10201.2**	3472.2	3982.4	**104.2**	**5189.5**	**-2.7**	**1000.0**	**9349.2**
	TD3 Paper	4372.4	9637.0	**3564.1**	**4682.8**	/	/	-3.6	**1000.0**	9337.5
	Spinning Up	~3800	~9750	~2860	~4000	~78	/	/	/	/
SAC	Tianshou	**5850.2**	**12138.8**	**3542.2**	**5007.0**	**44.4**	**5488.5**	**-2.6**	**1000.0**	**9359.5**
	SAC Paper	~3720	~10400	~3370	~3740	/	~5200	/	/	/
	TD3 Paper	655.4	2347.2	2996.7	1283.7	/	/	-4.4	**1000.0**	8487.2
	Spinning Up	~3980	~11520	~3150	~4250	~41.7	/	/	/	/
A2C	Tianshou	**3485.4**	**1829.9**	**1253.2**	**1091.6**	**36.6**	**1726.0**	**-6.7**	**1000.0**	**9257.7**
	PPO Paper	/	~1000	~900	~850	~31	/	~-24	**~1000**	~7100
	PPO Paper (TR)	/	~930	~1220	~700	**~36**	/	~-27	**~1000**	~8100
PPO	Tianshou	**3258.4**	**5783.9**	**2609.3**	3588.5	66.7	**787.1**	**-4.1**	**1000.0**	**9231.3**
	PPO Paper	/	~1800	~2330	~3460	~108	/	~-7	**~1000**	~8000
	TD3 Paper	1083.2	1795.4	2164.7	3317.7	/	/	-6.2	**1000.0**	8977.9
	OpenAI Baselines	/	~1700	~2400	~3510	~111	/	~-6	~940	~7350
	Spinning Up	~650	~1670	~1850	~1230	**~120**	/	/	/	/
TRPO	Tianshou	**2866.7**	**4471.2**	2046.0	**3826.7**	40.9	**810.1**	**-5.1**	**1000.0**	**8435.2**
	ACKTR paper	~0	~400	~1400	~550	~40	/	-8	**~1000**	~800
	PPO Paper	/	~0	~2100	~1100	**~121**	/	~-115	**~1000**	~200
	TD3 paper	-75.9	-15.6	**2471.3**	2321.5	/	/	-111.4	985.4	205.9
	OpenAI Baselines	/	~1350	**~2200**	~2350	~95	/	**~-5**	~910	~7000
	Spinning Up (TF)	~150	~850	~1200	~600	~85	/	/	/	/

Runtime averaged on 8 MuJoCo benchmark tasks is listed below. All results are obtained using a single Nvidia TITAN X GPU and up to 48 CPU cores (at most one CPU core for each thread).

Algorithm	# of Envs	1M timesteps	Collecting (%)	Updating (%)	Evaluating (%)	Others (%)
DDPG	1	2.9h	12.0	80.2	2.4	5.4
TD3	1	3.3h	11.4	81.7	1.7	5.2
SAC	1	5.2h	10.9	83.8	1.8	3.5
REINFORCE	64	4min	84.9	1.8	12.5	0.8
A2C	16	7min	62.5	28.0	6.6	2.9
PPO	64	24min	11.4	85.3	3.2	0.2
NPG	16	7min	65.1	24.9	9.5	0.6
TRPO	16	7min	62.9	26.5	10.1	0.6

1.7.2 Atari Benchmark

Tianshou also provides reliable and reproducible Atari 10M benchmark.

Every experiment is conducted under 10 random seeds for 10M steps. Please refer to https://github.com/thu-ml/tianshou/tree/master/examples/atari for source code and refer to https://wandb.ai/tianshou/atari.benchmark/reports/Atari-Benchmark–VmlldzoxOTA1NzA5 for detailed results hosted on wandb.

The table below compares the performance of Tianshou against published results on Atari games. We use max average return in 10M timesteps as the reward metric **(to be consistent with Mujoco)**. / means results are not provided. The best-performing baseline on each task is highlighted in boldface. Referenced baselines include Google Dopamine and OpenAI Baselines.

Task		Pong	Breakout	Enduro	Qbert	MsPacman	Seaquest	SpaceInvaders
DQN	Tianshou	**20.2 ± 2.3**	**133.5 ± 44.6**	997.9 ± 180.6	**11620.2 ± 786.1**	2324.8 ± 359.8	**3213.9 ± 381.6**	947.9 ± 155.3
	Dopamine	9.8	92.2	**2126.9**	6836.7	**2451.3**	1406.6	**1559.1**
	OpenAI Baselines	16.5	131.5	479.8	3254.8	/	1164.1	1129.5 ± 145.3
C51	Tianshou	**20.6 ± 2.4**	**412.9 ± 35.8**	**940.8 ± 133.9**	**12513.2 ± 1274.6**	2254.9 ± 201.2	**3305.4 ± 1524.3**	557.3
	Dopamine	17.4	222.4	665.3	9924.5	**2860.4**	1706.6	**604.6 ± 157.5**
Rainbow	Tianshou	**20.2 ± 3.0**	**440.4 ± 50.1**	1496.1 ± 112.3	14224.8 ± 1230.1	2524.2 ± 338.8	1934.6 ± 376.4	**1178.4**
	Dopamine	19.1	47.9	**2185.1**	**15682.2**	**3161.7**	**3328.9**	459.9
IQN	Tianshou	**20.7 ± 2.9**	**355.9 ± 22.7**	**1252.7 ± 118.1**	**14409.2 ± 808.6**	2228.6 ± 253.1	5341.2 ± 670.2	667.8 ± 81.5
	Dopamine	19.6	96.3	1227.6	12496.7	**4422.7**	**16418**	**1358.2 ± 267.6**
PPO	Tianshou	**20.3 ± 1.2**	**283.0 ± 74.3**	**1098.9 ± 110.5**	**12341.8 ± 1760.7**	1699.4 ± 248.0	1035.2 ± 353.6	1641.3
	OpenAI Baselines	13.7	114.3	350.2	7012.1	/	**1218.9**	**1787.5 ± 340.8**
QR-DQN	Tianshou	20.7 ± 2.0	228.3 ± 27.3	951.7 ± 333.5	14761.5 ± 862.9	2259.3 ± 269.2	4187.6 ± 725.7	1114.7 ± 116.9
FQF	Tianshou	20.4 ± 2.5	382.6 ± 29.5	1816.8 ± 314.3	15301.2 ± 684.1	2506.6 ± 402.5	8051.5 ± 3155.6	2558.3

Please note that the comparison table for both two benchmarks could NOT be used to prove which implementation is "better". The hyperparameters of the algorithms vary across different implementations. Also, the reward metric is not strictly the same (e.g. Tianshou uses max average return in 10M steps but OpenAI Baselines only report average return at 10M steps, which is unfair). Lastly, Tianshou always uses 10 random seeds while others might use fewer. The comparison is here only to show Tianshou's reliability.

1.8 Cheat Sheet

This page shows some code snippets of how to use Tianshou to develop new algorithms / apply algorithms to new scenarios.

By the way, some of these issues can be resolved by using a `gymnasium.Wrapper`. It could be a universal solution in the policy-environment interaction. But you can also use the batch processor *Handle Batched Data Stream in Collector* or vectorized environment wrapper *VectorEnvWrapper*.

1.8.1 Build Policy Network

See *Build the Network*.

1.8.2 Build New Policy

See *BasePolicy*.

1.8.3 Manually Evaluate Policy

If you'd like to manually see the action generated by a well-trained agent:

```
# assume obs is a single environment observation
action = policy(Batch(obs=np.array([obs]))).act[0]
```

1.8.4 Customize Training Process

See *Train a Policy with Customized Codes*.

1.8.5 Resume Training Process

This is related to Issue 349.

To resume training process from an existing checkpoint, you need to do the following things in the training process:

1. Make sure you write `save_checkpoint_fn` which saves everything needed in the training process, i.e., policy, optim, buffer; pass it to trainer;
2. Use `TensorboardLogger`;
3. To adjust the save frequency, specify `save_interval` when initializing TensorboardLogger.

And to successfully resume from a checkpoint:

1. Load everything needed in the training process **before trainer initialization**, i.e., policy, optim, buffer;
2. Set `resume_from_log=True` with trainer;

We provide an example to show how these steps work: checkout test_c51.py, test_ppo.py or test_discrete_bcq.py by running

```
$ python3 test/discrete/test_c51.py  # train some epoch
$ python3 test/discrete/test_c51.py --resume  # restore from existing log and continuing␣
↪training
```

To correctly render the data (including several tfevent files), we highly recommend using `tensorboard >= 2.5.0` (see here for the reason). Otherwise, it may cause overlapping issue that you need to manually handle with.

1.8.6 Parallel Sampling

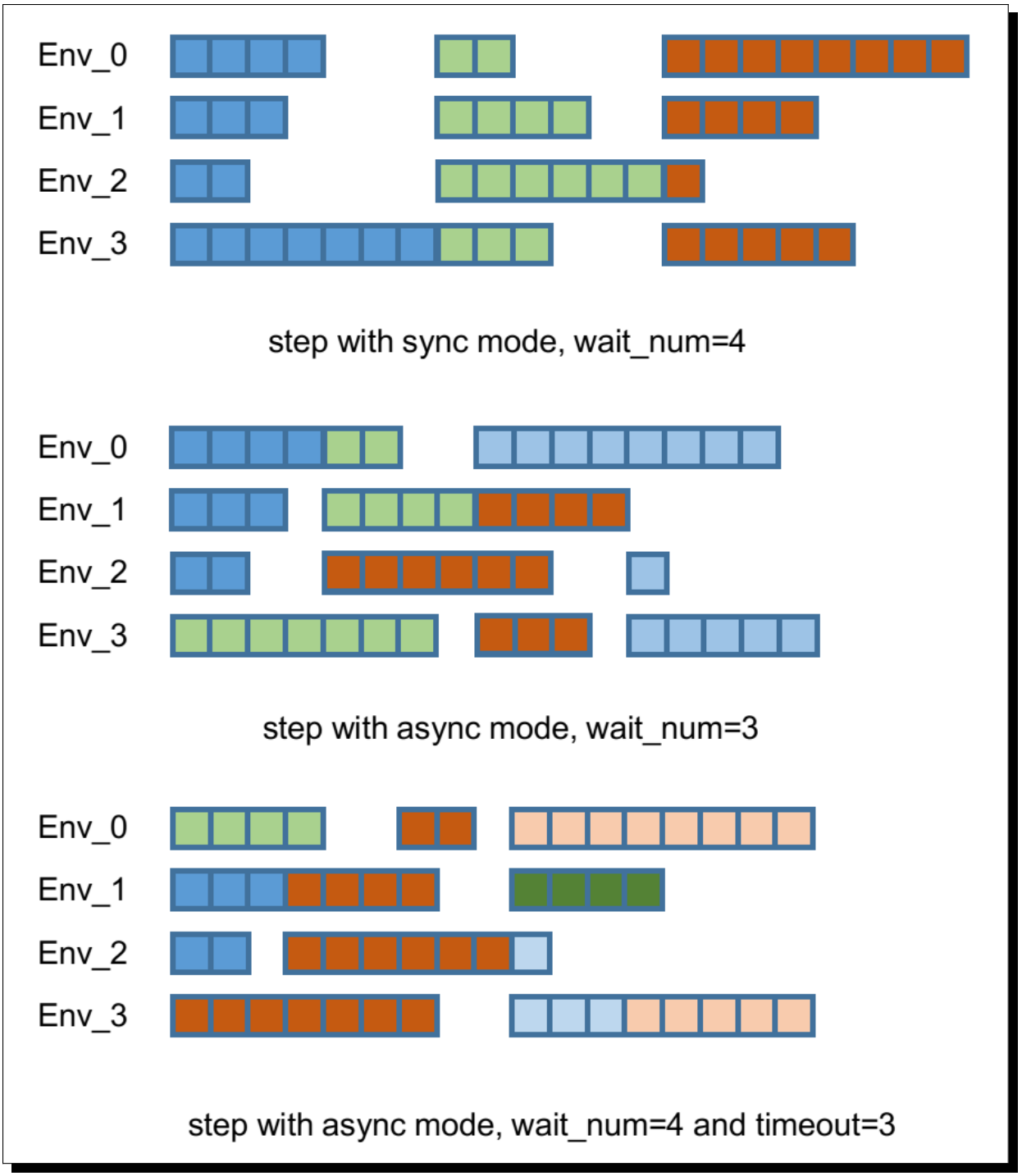
Tianshou provides the following classes for vectorized environment:

- *DummyVectorEnv* is for pseudo-parallel simulation (implemented with a for-loop, useful for debugging).
- *SubprocVectorEnv* uses multiple processes for parallel simulation. This is the most often choice for parallel simulation.
- *ShmemVectorEnv* has a similar implementation to *SubprocVectorEnv*, but is optimized (in terms of both memory footprint and simulation speed) for environments with large observations such as images.
- *RayVectorEnv* is currently the only choice for parallel simulation in a cluster with multiple machines.

Although these classes are optimized for different scenarios, they have exactly the same APIs because they are subclasses of *BaseVectorEnv*. Just provide a list of functions who return environments upon called, and it is all set.

```
env_fns = [lambda x=i: MyTestEnv(size=x) for i in [2, 3, 4, 5]]
venv = SubprocVectorEnv(env_fns)  # DummyVectorEnv, ShmemVectorEnv, or RayVectorEnv,␣
↪whichever you like.
venv.reset()  # returns the initial observations of each environment
venv.step(actions)  # provide actions for each environment and get their results
```

An example of sync/async VectorEnv (steps with the same color end up in one batch that is disposed by the policy at the same time).



By default, parallel environment simulation is synchronous: a step is done after all environments have finished a step. Synchronous simulation works well if each step of environments costs roughly the same time.

In case the time cost of environments varies a lot (e.g. 90% step cost 1s, but 10% cost 10s) where slow environments lag fast environments behind, async simulation can be used (related to Issue 103). The idea is to start those finished environments without waiting for slow environments.

Asynchronous simulation is a built-in functionality of *BaseVectorEnv*. Just provide `wait_num` or `timeout` (or both) and async simulation works.

```
env_fns = [lambda x=i: MyTestEnv(size=x, sleep=x) for i in [2, 3, 4, 5]]
# DummyVectorEnv, ShmemVectorEnv, or RayVectorEnv, whichever you like.
venv = SubprocVectorEnv(env_fns, wait_num=3, timeout=0.2)
venv.reset()  # returns the initial observations of each environment
# returns "wait_num" steps or finished steps after "timeout" seconds,
# whichever occurs first.
venv.step(actions, ready_id)
```

If we have 4 envs and set `wait_num = 3`, each of the step only returns 3 results of these 4 envs.

You can treat the `timeout` parameter as a dynamic `wait_num`. In each vectorized step it only returns the environments finished within the given time. If there is no such environment, it will wait until any of them finished.

The figure in the right gives an intuitive comparison among synchronous/asynchronous simulation.

Note: The async simulation collector would cause some exceptions when used as `test_collector` in *tianshou.trainer* (related to Issue 700). Please use sync version for `test_collector` instead.

Warning: If you use your own environment, please make sure the `seed` method is set up properly, e.g.,

```
def seed(self, seed):
    np.random.seed(seed)
```

Otherwise, the outputs of these envs may be the same with each other.

1.8.7 EnvPool Integration

EnvPool is a C++-based vectorized environment implementation and is way faster than the above solutions. The APIs are almost the same as above four classes, so that means you can directly switch the vectorized environment to envpool and get immediate speed-up.

Currently it supports Atari, Mujoco, VizDoom, toy_text and classic_control environments. For more information, please refer to EnvPool's documentation.

```
# install envpool: pip3 install envpool

import envpool
envs = envpool.make_gymnasium("CartPole-v0", num_envs=10)
collector = Collector(policy, envs, buffer)
```

Here are some other examples.

1.8.8 Handle Batched Data Stream in Collector

This is related to Issue 42.

If you want to get log stat from data stream / pre-process batch-image / modify the reward with given env info, use `preproces_fn` in *Collector*. This is a hook which will be called before the data adding into the buffer.

It will receive with "obs" and "env_id" when the collector resets the environment, and will receive six keys "obs_next", "rew", "done", "info", "policy", "env_id" in a normal env step. It returns either a dict or a *Batch* with the modified keys and values.

These variables are intended to gather all the information requires to keep track of a simulation step, namely the (observation, action, reward, done flag, next observation, info, intermediate result of the policy) at time t, for the whole duration of the simulation.

For example, you can write your hook as:

```
import numpy as np
from collections import deque


class MyProcessor:
    def __init__(self, size=100):
        self.episode_log = None
        self.main_log = deque(maxlen=size)
        self.main_log.append(0)
        self.baseline = 0

    def preprocess_fn(**kwargs):
        """change reward to zero mean"""
        # if obs && env_id exist -> reset
        # if obs_next/act/rew/done/policy/env_id exist -> normal step
        if 'rew' not in kwargs:
            # means that it is called after env.reset(), it can only process the obs
            return Batch()  # none of the variables are needed to be updated
        else:
            n = len(kwargs['rew'])  # the number of envs in collector
            if self.episode_log is None:
                self.episode_log = [[] for i in range(n)]
            for i in range(n):
                self.episode_log[i].append(kwargs['rew'][i])
                kwargs['rew'][i] -= self.baseline
            for i in range(n):
                if kwargs['done'][i]:
                    self.main_log.append(np.mean(self.episode_log[i]))
                    self.episode_log[i] = []
                    self.baseline = np.mean(self.main_log)
            return Batch(rew=kwargs['rew'])
```

And finally,

```
test_processor = MyProcessor(size=100)
collector = Collector(policy, env, buffer, preprocess_fn=test_processor.preprocess_fn)
```

Some examples are in test/base/test_collector.py.

Another solution is to create a vector environment wrapper through *VectorEnvWrapper*, e.g.

```
import numpy as np
from collections import deque
from tianshou.env import VectorEnvWrapper

class MyWrapper(VectorEnvWrapper):
    def __init__(self, venv, size=100):
        self.episode_log = None
        self.main_log = deque(maxlen=size)
        self.main_log.append(0)
        self.baseline = 0

    def step(self, action, env_id):
        obs, rew, done, info = self.venv.step(action, env_id)
        n = len(rew)
        if self.episode_log is None:
            self.episode_log = [[] for i in range(n)]
        for i in range(n):
            self.episode_log[i].append(rew[i])
            rew[i] -= self.baseline
        for i in range(n):
            if done[i]:
                self.main_log.append(np.mean(self.episode_log[i]))
                self.episode_log[i] = []
                self.baseline = np.mean(self.main_log)
        return obs, rew, done, info

env = MyWrapper(env, size=100)
collector = Collector(policy, env, buffer)
```

We provide an observation normalization vector env wrapper: *VectorEnvNormObs*.

1.8.9 RNN-style Training

This is related to Issue 19.

First, add an argument "stack_num" to *ReplayBuffer*, *VectorReplayBuffer*, or other types of buffer you are using, like:

```
buf = ReplayBuffer(size=size, stack_num=stack_num)
```

Then, change the network to recurrent-style, for example, *Recurrent*, *RecurrentActorProb* and *RecurrentCritic*.

The above code supports only stacked-observation. If you want to use stacked-action (for Q(stacked-s, stacked-a)), stacked-reward, or other stacked variables, you can add a `gym.Wrapper` to modify the state representation. For example, if we add a wrapper that map `[s, a]` pair to a new state:

- Before: `(s, a, s', r, d)` stored in replay buffer, and get stacked s;
- After applying wrapper: `([s, a], a, [s', a'], r, d)` stored in replay buffer, and get both stacked s and a.

1.8.10 Multi-GPU Training

To enable training an RL agent with multiple GPUs for a standard environment (i.e., without nested observation) with default networks provided by Tianshou:

1. Import *DataParallelNet* from `tianshou.utils.net.common`;
2. Change the `device` argument to `None` in the existing networks such as `Net`, `Actor`, `Critic`, `ActorProb`
3. Apply `DataParallelNet` wrapper to these networks.

```
from tianshou.utils.net.common import Net, DataParallelNet
from tianshou.utils.net.discrete import Actor, Critic

actor = DataParallelNet(Actor(net, args.action_shape, device=None).to(args.device))
critic = DataParallelNet(Critic(net, device=None).to(args.device))
```

Yes, that's all! This general approach can be applied to almost all kinds of algorithms implemented in Tianshou. We provide a complete script to show how to run multi-GPU: test/discrete/test_ppo.py

As for other cases such as customized network or environments that have a nested observation, here are the rules:

1. The data format transformation (numpy -> cuda) is done in the `DataParallelNet` wrapper; your customized network should not apply any kinds of data format transformation;
2. Create a similar class that inherit `DataParallelNet`, which is only in charge of data format transformation (numpy -> cuda);
3. Do the same things above.

1.8.11 User-defined Environment and Different State Representation

This is related to Issue 38 and Issue 69.

First of all, your self-defined environment must follow the Gym's API, some of them are listed below:

- reset() -> state
- step(action) -> state, reward, done, info
- seed(s) -> List[int]
- render(mode) -> Any
- close() -> None
- observation_space: gym.Space
- action_space: gym.Space

The state can be a `numpy.ndarray` or a Python dictionary. Take "FetchReach-v1" as an example:

```
>>> e = gym.make('FetchReach-v1')
>>> e.reset()
{'observation': array([ 1.34183265e+00,  7.49100387e-01,  5.34722720e-01,  1.97805133e-
↪04,
         7.15193042e-05,  7.73933014e-06,  5.51992816e-08, -2.42927453e-06,
         4.73325650e-06, -2.28455228e-06]),
 'achieved_goal': array([1.34183265, 0.74910039, 0.53472272]),
 'desired_goal': array([1.24073906, 0.77753463, 0.63457791])}
```

It shows that the state is a dictionary which has 3 keys. It will stored in *ReplayBuffer* as:

```
>>> from tianshou.data import Batch, ReplayBuffer
>>> b = ReplayBuffer(size=3)
>>> b.add(Batch(obs=e.reset(), act=0, rew=0, done=0))
>>> print(b)
ReplayBuffer(
    act: array([0, 0, 0]),
    done: array([False, False, False]),
    obs: Batch(
             achieved_goal: array([[1.34183265, 0.74910039, 0.53472272],
                                   [0.        , 0.        , 0.        ],
                                   [0.        , 0.        , 0.        ]]),
             desired_goal: array([[1.42154265, 0.62505137, 0.62929863],
                                  [0.        , 0.        , 0.        ],
                                  [0.        , 0.        , 0.        ]]),
             observation: array([[ 1.34183265e+00,  7.49100387e-01,  5.34722720e-01,
                                   1.97805133e-04,  7.15193042e-05,  7.73933014e-06,
                                   5.51992816e-08, -2.42927453e-06,  4.73325650e-06,
                                  -2.28455228e-06],
                                 [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
                                   0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
                                   0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
                                   0.00000000e+00],
                                 [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
                                   0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
                                   0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
                                   0.00000000e+00]]),
         ),
    rew: array([0, 0, 0]),
)
>>> print(b.obs.achieved_goal)
[[1.34183265 0.74910039 0.53472272]
 [0.         0.         0.        ]
 [0.         0.         0.        ]]
```

And the data batch sampled from this replay buffer:

```
>>> batch, indices = b.sample(2)
>>> batch.keys()
['act', 'done', 'info', 'obs', 'obs_next', 'policy', 'rew']
>>> batch.obs[-1]
Batch(
    achieved_goal: array([1.34183265, 0.74910039, 0.53472272]),
    desired_goal: array([1.42154265, 0.62505137, 0.62929863]),
    observation: array([ 1.34183265e+00,  7.49100387e-01,  5.34722720e-01,  1.97805133e-
↪04,
                         7.15193042e-05,  7.73933014e-06,  5.51992816e-08, -2.42927453e-
↪06,
                         4.73325650e-06, -2.28455228e-06]),
)
>>> batch.obs.desired_goal[-1]  # recommended
array([1.42154265, 0.62505137, 0.62929863])
>>> batch.obs[-1].desired_goal  # not recommended
```

(continues on next page)

(continued from previous page)

```
array([1.42154265, 0.62505137, 0.62929863])
>>> batch[-1].obs.desired_goal  # not recommended
array([1.42154265, 0.62505137, 0.62929863])
```

Thus, in your self-defined network, just change the `forward` function as:

```
def forward(self, s, ...):
    # s is a batch
    observation = s.observation
    achieved_goal = s.achieved_goal
    desired_goal = s.desired_goal
    ...
```

For self-defined class, the replay buffer will store the reference into a `numpy.ndarray`, e.g.:

```
>>> import networkx as nx
>>> b = ReplayBuffer(size=3)
>>> b.add(Batch(obs=nx.Graph(), act=0, rew=0, done=0))
>>> print(b)
ReplayBuffer(
    act: array([0, 0, 0]),
    done: array([0, 0, 0]),
    info: Batch(),
    obs: array([<networkx.classes.graph.Graph object at 0x7f5c607826a0>, None,
                None], dtype=object),
    policy: Batch(),
    rew: array([0, 0, 0]),
)
```

But the state stored in the buffer may be a shallow-copy. To make sure each of your state stored in the buffer is distinct, please return the deep-copy version of your state in your env:

```
def reset():
    return copy.deepcopy(self.graph)
def step(action):
    ...
    return copy.deepcopy(self.graph), reward, done, {}
```

Note: Please make sure this variable is numpy-compatible, e.g., np.array([variable]) will not result in an empty array. Otherwise, ReplayBuffer cannot create an numpy array to store it.

1.8.12 Multi-Agent Reinforcement Learning

This is related to Issue 121. The discussion is still goes on.

With the flexible core APIs, Tianshou can support multi-agent reinforcement learning with minimal efforts.

Currently, we support three types of multi-agent reinforcement learning paradigms:

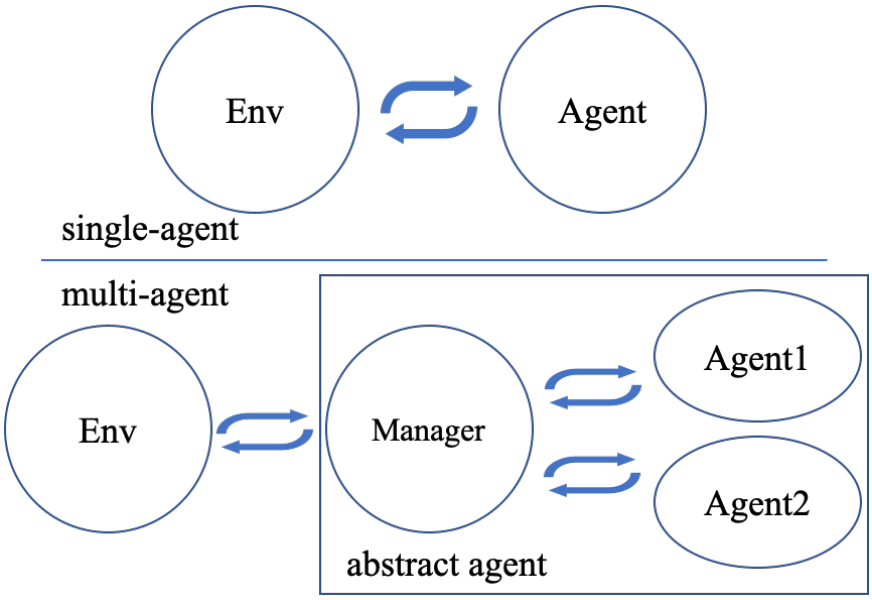
1. Simultaneous move: at each timestep, all the agents take their actions (example: MOBA games)
2. Cyclic move: players take action in turn (example: Go game)
3. Conditional move, at each timestep, the environment conditionally selects an agent to take action. (example: Pig Game)

We mainly address these multi-agent RL problems by converting them into traditional RL formulations.

For simultaneous move, the solution is simple: we can just add a `num_agent` dimension to state, action, and reward. Nothing else is going to change.

For 2 & 3 (cyclic move and conditional move), they can be unified into a single framework: at each timestep, the environment selects an agent with id `agent_id` to play. Since multi-agents are usually wrapped into one object (which we call "abstract agent"), we can pass the `agent_id` to the "abstract agent", leaving it to further call the specific agent.

In addition, legal actions in multi-agent RL often vary with timestep (just like Go games), so the environment should also passes the legal action mask to the "abstract agent", where the mask is a boolean array that "True" for available actions and "False" for illegal actions at the current step. Below is a figure that explains the abstract agent.



The above description gives rise to the following formulation of multi-agent RL:

```
act = policy(state, agent_id, mask)
(next_state, next_agent_id, next_mask), reward = env.step(act)
```

By constructing a new state `state_ = (state, agent_id, mask)`, essentially we can return to the typical formulation of RL:

```
act = policy(state_)
next_state_, reward = env.step(act)
```

Following this idea, we write a tiny example of playing Tic Tac Toe against a random player by using a Q-learning algorithm. The tutorial is at *Multi-Agent RL*.

1.9 tianshou.data

1.9.1 Batch

class tianshou.data.**Batch**(*batch_dict: Optional[Union[dict, Batch, Sequence[Union[dict, Batch]], ndarray]] = None*, *copy: bool = False*, ***kwargs: Any*)

Bases: `object`

The internal data structure in Tianshou.

Batch is a kind of supercharged array (of temporal data) stored individually in a (recursive) dictionary of object that can be either numpy array, torch tensor, or batch themselves. It is designed to make it extremely easily to access, manipulate and set partial view of the heterogeneous data conveniently.

For a detailed description, please refer to *Understand Batch*.

__getitem__(*index: Union[str, slice, int, ndarray, List[int]]*) → Any

Return self[index].

__setitem__(*index: Union[str, slice, int, ndarray, List[int]]*, *value: Any*) → None

Assign value to self[index].

to_numpy() → None

Change all torch.Tensor to numpy.ndarray in-place.

to_torch(*dtype: Optional[dtype] = None*, *device: Union[str, int, device] = 'cpu'*) → None

Change all numpy.ndarray to torch.Tensor in-place.

cat_(*batches: Union[Batch, Sequence[Union[dict, Batch]]]*) → None

Concatenate a list of (or one) Batch objects into current batch.

static cat(*batches: Sequence[Union[dict, Batch]]*) → *Batch*

Concatenate a list of Batch object into a single new batch.

For keys that are not shared across all batches, batches that do not have these keys will be padded by zeros with appropriate shapes. E.g.

```
>>> a = Batch(a=np.zeros([3, 4]), common=Batch(c=np.zeros([3, 5])))
>>> b = Batch(b=np.zeros([4, 3]), common=Batch(c=np.zeros([4, 5])))
>>> c = Batch.cat([a, b])
>>> c.a.shape
(7, 4)
>>> c.b.shape
(7, 3)
>>> c.common.c.shape
(7, 5)
```

stack_(*batches: Sequence[Union[dict, Batch]]*, *axis: int = 0*) → None

Stack a list of Batch object into current batch.

static stack(*batches: Sequence[Union[dict, Batch]]*, *axis: int = 0*) → *Batch*

Stack a list of Batch object into a single new batch.

For keys that are not shared across all batches, batches that do not have these keys will be padded by zeros. E.g.

```
>>> a = Batch(a=np.zeros([4, 4]), common=Batch(c=np.zeros([4, 5])))
>>> b = Batch(b=np.zeros([4, 6]), common=Batch(c=np.zeros([4, 5])))
>>> c = Batch.stack([a, b])
>>> c.a.shape
(2, 4, 4)
>>> c.b.shape
(2, 4, 6)
>>> c.common.c.shape
(2, 4, 5)
```

Note: If there are keys that are not shared across all batches, `stack` with `axis != 0` is undefined, and will cause an exception.

empty_(*index: Optional[Union[slice, int, ndarray, List[int]]] = None*) → *Batch*

Return an empty Batch object with 0 or None filled.

If "index" is specified, it will only reset the specific indexed-data.

```
>>> data.empty_()
>>> print(data)
Batch(
    a: array([[0., 0.],
              [0., 0.]]),
    b: array([None, None], dtype=object),
)
>>> b={'c': [2., 'st'], 'd': [1., 0.]}
>>> data = Batch(a=[False,  True], b=b)
>>> data[0] = Batch.empty(data[1])
>>> data
Batch(
    a: array([False,  True]),
    b: Batch(
           c: array([None, 'st']),
           d: array([0., 0.]),
       ),
)
```

static empty(*batch:* Batch, *index: Optional[Union[slice, int, ndarray, List[int]]] = None*) → *Batch*

Return an empty Batch object with 0 or None filled.

The shape is the same as the given Batch.

update(*batch: Optional[Union[dict,* Batch*]] = None*, ***kwargs: Any*) → None

Update this batch from another dict/Batch.

__len__() → int

Return len(self).

is_empty(*recurse: bool = False*) → bool

Test if a Batch is empty.

If `recurse=True`, it further tests the values of the object; else it only tests the existence of any key.

`b.is_empty(recurse=True)` is mainly used to distinguish `Batch(a=Batch(a=Batch()))` and `Batch(a=1)`. They both raise exceptions when applied to `len()`, but the former can be used in `cat`, while the latter is a scalar and cannot be used in `cat`.

Another usage is in `__len__`, where we have to skip checking the length of recursively empty Batch.

```
>>> Batch().is_empty()
True
>>> Batch(a=Batch(), b=Batch(c=Batch())).is_empty()
False
>>> Batch(a=Batch(), b=Batch(c=Batch())).is_empty(recurse=True)
True
>>> Batch(d=1).is_empty()
False
>>> Batch(a=np.float64(1.0)).is_empty()
False
```

property shape: List[int]

Return self.shape.

split(*size: int*, *shuffle: bool = True*, *merge_last: bool = False*) → Iterator[*Batch*]

Split whole data into multiple small batches.

Parameters

- **size** (*int*) – divide the data batch with the given size, but one batch if the length of the batch is smaller than "size".
- **shuffle** (*bool*) – randomly shuffle the entire data batch if it is True, otherwise remain in the same. Default to True.
- **merge_last** (*bool*) – merge the last batch into the previous one. Default to False.

1.9.2 Buffer

ReplayBuffer

class tianshou.data.**ReplayBuffer**(*size: int*, *stack_num: int = 1*, *ignore_obs_next: bool = False*, *save_only_last_obs: bool = False*, *sample_avail: bool = False*, ***kwargs: Any*)

Bases: `object`

ReplayBuffer stores data generated from interaction between the policy and environment.

ReplayBuffer can be considered as a specialized form (or management) of Batch. It stores all the data in a batch with circular-queue style.

For the example usage of ReplayBuffer, please check out Section Buffer in *Basic concepts in Tianshou*.

Parameters

- **size** (*int*) – the maximum size of replay buffer.

- **stack_num** (*int*) – the frame-stack sampling argument, should be greater than or equal to 1. Default to 1 (no stacking).
- **ignore_obs_next** (*bool*) – whether to store obs_next. Default to False.
- **save_only_last_obs** (*bool*) – only save the last obs/obs_next when it has a shape of (timestep, ...) because of temporal stacking. Default to False.
- **sample_avail** (*bool*) – the parameter indicating sampling only available index when using frame-stack sampling method. Default to False.

__len__() → int

Return len(self).

save_hdf5(*path: str*, *compression: Optional[str] = None*) → None

Save replay buffer to HDF5 file.

classmethod load_hdf5(*path: str*, *device: Optional[str] = None*) → *ReplayBuffer*

Load replay buffer from HDF5 file.

classmethod from_data(*obs: Dataset*, *act: Dataset*, *rew: Dataset*, *terminated: Dataset*, *truncated: Dataset*, *done: Dataset*, *obs_next: Dataset*) → *ReplayBuffer*

reset(*keep_statistics: bool = False*) → None

Clear all the data in replay buffer and episode statistics.

set_batch(*batch:* Batch) → None

Manually choose the batch you want the ReplayBuffer to manage.

unfinished_index() → ndarray

Return the index of unfinished episode.

prev(*index: Union[int, ndarray]*) → ndarray

Return the index of previous transition.

The index won't be modified if it is the beginning of an episode.

next(*index: Union[int, ndarray]*) → ndarray

Return the index of next transition.

The index won't be modified if it is the end of an episode.

update(*buffer:* ReplayBuffer) → ndarray

Move the data from the given buffer to current buffer.

Return the updated indices. If update fails, return an empty array.

add(*batch:* Batch, *buffer_ids: Optional[Union[ndarray, List[int]]] = None*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Add a batch of data into replay buffer.

Parameters

- **batch** (Batch) – the input data batch. Its keys must belong to the 7 input keys, and "obs", "act", "rew", "terminated", "truncated" is required.
- **buffer_ids** – to make consistent with other buffer's add function; if it is not None, we assume the input batch's first dimension is always 1.

Return (current_index, episode_reward, episode_length, episode_start_index). If the episode is not finished, the return value of episode_length and episode_reward is 0.

sample_indices(*batch_size: int*) → ndarray

Get a random sample of index with size = batch_size.

Return all available indices in the buffer if batch_size is 0; return an empty numpy array if batch_size < 0 or no available index can be sampled.

sample(*batch_size: int*) → Tuple[*Batch*, ndarray]

Get a random sample from buffer with size = batch_size.

Return all the data in the buffer if batch_size is 0.

Returns

Sample data and its corresponding index inside the buffer.

get(*index: Union[int, List[int], ndarray]*, *key: str*, *default_value: Optional[Any] = None*, *stack_num: Optional[int] = None*) → Union[*Batch*, ndarray]

Return the stacked result.

E.g., if you set `key = "obs", stack_num = 4, index = t`, it returns the stacked result as `[obs[t-3], obs[t-2], obs[t-1], obs[t]]`.

Parameters

- **index** – the index for getting stacked data.
- **key** (*str*) – the key to get, should be one of the reserved_keys.
- **default_value** – if the given key's data is not found and default_value is set, return this default_value.
- **stack_num** (*int*) – Default to self.stack_num.

__getitem__(*index: Union[slice, int, List[int], ndarray]*) → *Batch*

Return a data batch: self[index].

If stack_num is larger than 1, return the stacked obs and obs_next with shape (batch, len, ...).

PrioritizedReplayBuffer

class tianshou.data.**PrioritizedReplayBuffer**(*size: int*, *alpha: float*, *beta: float*, *weight_norm: bool = True*, ***kwargs: Any*)

Bases: *ReplayBuffer*

Implementation of Prioritized Experience Replay. arXiv:1511.05952.

Parameters

- **alpha** (*float*) – the prioritization exponent.
- **beta** (*float*) – the importance sample soft coefficient.
- **weight_norm** (*bool*) – whether to normalize returned weights with the maximum weight value within the batch. Default to True.

See also:

Please refer to *ReplayBuffer* for other APIs' usage.

init_weight(*index: Union[int, ndarray]*) → None

update(*buffer:* ReplayBuffer) → ndarray

Move the data from the given buffer to current buffer.

Return the updated indices. If update fails, return an empty array.

add(*batch:* Batch, *buffer_ids: Optional[Union[ndarray, List[int]]] = None*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Add a batch of data into replay buffer.

Parameters

- **batch** (Batch) – the input data batch. Its keys must belong to the 7 input keys, and "obs", "act", "rew", "terminated", "truncated" is required.
- **buffer_ids** – to make consistent with other buffer's add function; if it is not None, we assume the input batch's first dimension is always 1.

Return (current_index, episode_reward, episode_length, episode_start_index). If the episode is not finished, the return value of episode_length and episode_reward is 0.

sample_indices(*batch_size: int*) → ndarray

Get a random sample of index with size = batch_size.

Return all available indices in the buffer if batch_size is 0; return an empty numpy array if batch_size < 0 or no available index can be sampled.

get_weight(*index: Union[int, ndarray]*) → Union[float, ndarray]

Get the importance sampling weight.

The "weight" in the returned Batch is the weight on loss function to debias the sampling process (some transition tuples are sampled more often so their losses are weighted less).

update_weight(*index: ndarray*, *new_weight: Union[ndarray, Tensor]*) → None

Update priority weight by index in this buffer.

Parameters

- **index** (*np.ndarray*) – index you want to update weight.
- **new_weight** (*np.ndarray*) – new priority weight you want to update.

__getitem__(*index: Union[slice, int, List[int], ndarray]*) → *Batch*

Return a data batch: self[index].

If stack_num is larger than 1, return the stacked obs and obs_next with shape (batch, len, ...).

set_beta(*beta: float*) → None

HERReplayBuffer

class tianshou.data.**HERReplayBuffer**(*size: int*, *compute_reward_fn: Callable[[ndarray, ndarray], ndarray]*, *horizon: int*, *future_k: float = 8.0*, ***kwargs: Any*)

Bases: *ReplayBuffer*

Implementation of Hindsight Experience Replay. arXiv:1707.01495.

HERReplayBuffer is to be used with goal-based environment where the observation is a dictionary with keys `observation`, `achieved_goal` and `desired_goal`. Currently support only HER's future strategy, online sampling.

Parameters

- **size** (*int*) – the size of the replay buffer.
- **compute_reward_fn** – a function that takes 2 `np.array` arguments, `acheived_goal` and `desired_goal`, and returns rewards as `np.array`. The two arguments are of shape (batch_size, ... original_shape) and the returned rewards must be of shape (batch_size,).
- **horizon** (*int*) – the maximum number of steps in an episode.
- **future_k** (*int*) – the 'k' parameter introduced in the paper. In short, there will be at most k episodes that are re-written for every 1 unaltered episode during the sampling.

See also:

Please refer to *ReplayBuffer* for other APIs' usage.

reset(*keep_statistics: bool = False*) → None

Clear all the data in replay buffer and episode statistics.

save_hdf5(*path: str*, *compression: Optional[str] = None*) → None

Save replay buffer to HDF5 file.

set_batch(*batch:* Batch) → None

Manually choose the batch you want the ReplayBuffer to manage.

update(*buffer: Union[*HERReplayBuffer, ReplayBuffer*]*) → ndarray

Move the data from the given buffer to current buffer.

Return the updated indices. If update fails, return an empty array.

add(*batch:* Batch, *buffer_ids: Optional[Union[ndarray, List[int]]] = None*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Add a batch of data into replay buffer.

Parameters

- **batch** (Batch) – the input data batch. Its keys must belong to the 7 input keys, and "obs", "act", "rew", "terminated", "truncated" is required.
- **buffer_ids** – to make consistent with other buffer's add function; if it is not None, we assume the input batch's first dimension is always 1.

Return (current_index, episode_reward, episode_length, episode_start_index). If the episode is not finished, the return value of episode_length and episode_reward is 0.

sample_indices(*batch_size: int*) → ndarray

Get a random sample of index with size = batch_size.

Return all available indices in the buffer if batch_size is 0; return an empty numpy array if batch_size < 0 or no available index can be sampled. Additionally, some episodes of the sampled transitions will be re-written according to HER.

rewrite_transitions(*indices: ndarray*) → None

Re-write the goal of some sampled transitions' episodes according to HER.

Currently applies only HER's 'future' strategy. The new goals will be written directly to the internal batch data temporarily and will be restored right before the next sampling or when using some of the buffer's method (e.g. *add*, *save_hdf5*, etc.). This is to make sure that n-step returns calculation etc., performs correctly without additional alteration.

ReplayBufferManager

class tianshou.data.**ReplayBufferManager**(*buffer_list: Union[List[*ReplayBuffer*], List[*HERReplayBuffer*]]*)

Bases: *ReplayBuffer*

ReplayBufferManager contains a list of ReplayBuffer with exactly the same configuration.

These replay buffers have contiguous memory layout, and the storage space each buffer has is a shallow copy of the topmost memory.

Parameters
buffer_list – a list of ReplayBuffer needed to be handled.

See also:

Please refer to *ReplayBuffer* for other APIs' usage.

__len__() → int

Return len(self).

reset(*keep_statistics: bool = False*) → None

Clear all the data in replay buffer and episode statistics.

set_batch(*batch:* Batch) → None

Manually choose the batch you want the ReplayBuffer to manage.

unfinished_index() → ndarray

Return the index of unfinished episode.

prev(*index: Union[int, ndarray]*) → ndarray

Return the index of previous transition.

The index won't be modified if it is the beginning of an episode.

next(*index: Union[int, ndarray]*) → ndarray

Return the index of next transition.

The index won't be modified if it is the end of an episode.

update(*buffer:* ReplayBuffer) → ndarray

The ReplayBufferManager cannot be updated by any buffer.

add(*batch:* Batch, *buffer_ids: Optional[Union[ndarray, List[int]]] = None*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Add a batch of data into ReplayBufferManager.

Each of the data's length (first dimension) must equal to the length of buffer_ids. By default buffer_ids is [0, 1, . . . , buffer_num - 1].

Return (current_index, episode_reward, episode_length, episode_start_index). If the episode is not finished, the return value of episode_length and episode_reward is 0.

sample_indices(*batch_size: int*) → ndarray

Get a random sample of index with size = batch_size.

Return all available indices in the buffer if batch_size is 0; return an empty numpy array if batch_size < 0 or no available index can be sampled.

PrioritizedReplayBufferManager

class tianshou.data.**PrioritizedReplayBufferManager**(*buffer_list: Sequence[*PrioritizedReplayBuffer*]*)

Bases: *PrioritizedReplayBuffer*, *ReplayBufferManager*

PrioritizedReplayBufferManager contains a list of PrioritizedReplayBuffer with exactly the same configuration.

These replay buffers have contiguous memory layout, and the storage space each buffer has is a shallow copy of the topmost memory.

Parameters
buffer_list – a list of PrioritizedReplayBuffer needed to be handled.

See also:

Please refer to *ReplayBuffer* for other APIs' usage.

HERReplayBufferManager

class tianshou.data.**HERReplayBufferManager**(*buffer_list: List[*HERReplayBuffer*]*)

Bases: *ReplayBufferManager*

HERReplayBufferManager contains a list of HERReplayBuffer with exactly the same configuration.

These replay buffers have contiguous memory layout, and the storage space each buffer has is a shallow copy of the topmost memory.

Parameters
buffer_list – a list of HERReplayBuffer needed to be handled.

See also:

Please refer to *ReplayBuffer* for other APIs' usage.

save_hdf5(*path: str*, *compression: Optional[str] = None*) → None

Save replay buffer to HDF5 file.

set_batch(*batch:* Batch) → None

Manually choose the batch you want the ReplayBuffer to manage.

update(*buffer: Union[*HERReplayBuffer*,* ReplayBuffer*]*) → ndarray

The ReplayBufferManager cannot be updated by any buffer.

add(*batch:* Batch, *buffer_ids: Optional[Union[ndarray, List[int]]] = None*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Add a batch of data into ReplayBufferManager.

Each of the data's length (first dimension) must equal to the length of buffer_ids. By default buffer_ids is [0, 1, ..., buffer_num - 1].

Return (current_index, episode_reward, episode_length, episode_start_index). If the episode is not finished, the return value of episode_length and episode_reward is 0.

VectorReplayBuffer

class tianshou.data.**VectorReplayBuffer**(*total_size: int*, *buffer_num: int*, ***kwargs: Any*)

Bases: *ReplayBufferManager*

VectorReplayBuffer contains n ReplayBuffer with the same size.

It is used for storing transition from different environments yet keeping the order of time.

Parameters

- **total_size** (*int*) – the total size of VectorReplayBuffer.
- **buffer_num** (*int*) – the number of ReplayBuffer it uses, which are under the same configuration.

Other input arguments (stack_num/ignore_obs_next/save_only_last_obs/sample_avail) are the same as *ReplayBuffer*.

See also:

Please refer to *ReplayBuffer* for other APIs' usage.

PrioritizedVectorReplayBuffer

class tianshou.data.**PrioritizedVectorReplayBuffer**(*total_size: int*, *buffer_num: int*, ***kwargs: Any*)

Bases: *PrioritizedReplayBufferManager*

PrioritizedVectorReplayBuffer contains n PrioritizedReplayBuffer with same size.

It is used for storing transition from different environments yet keeping the order of time.

Parameters

- **total_size** (*int*) – the total size of PrioritizedVectorReplayBuffer.
- **buffer_num** (*int*) – the number of PrioritizedReplayBuffer it uses, which are under the same configuration.

Other input arguments (alpha/beta/stack_num/ignore_obs_next/save_only_last_obs/ sample_avail) are the same as *PrioritizedReplayBuffer*.

See also:

Please refer to *ReplayBuffer* for other APIs' usage.

set_beta(*beta: float*) → None

HERVectorReplayBuffer

class tianshou.data.**HERVectorReplayBuffer**(*total_size: int*, *buffer_num: int*, ***kwargs: Any*)

Bases: *HERReplayBufferManager*

HERVectorReplayBuffer contains n HERReplayBuffer with same size.

It is used for storing transition from different environments yet keeping the order of time.

Parameters

- **total_size** (*int*) – the total size of HERVectorReplayBuffer.
- **buffer_num** (*int*) – the number of HERReplayBuffer it uses, which are under the same configuration.

Other input arguments are the same as *HERReplayBuffer*.

See also:

Please refer to *ReplayBuffer* for other APIs' usage.

CachedReplayBuffer

class tianshou.data.**CachedReplayBuffer**(*main_buffer: ReplayBuffer*, *cached_buffer_num: int*, *max_episode_length: int*)

Bases: *ReplayBufferManager*

CachedReplayBuffer contains a given main buffer and n cached buffers, `cached_buffer_num * ReplayBuffer(size=max_episode_length)`.

The memory layout is: `| main_buffer | cached_buffers[0] | cached_buffers[1] | ... | cached_buffers[cached_buffer_num - 1] |`.

The data is first stored in cached buffers. When an episode is terminated, the data will move to the main buffer and the corresponding cached buffer will be reset.

Parameters

- **main_buffer** (ReplayBuffer) – the main buffer whose `.update()` function behaves normally.
- **cached_buffer_num** (*int*) – number of ReplayBuffer needs to be created for cached buffer.
- **max_episode_length** (*int*) – the maximum length of one episode, used in each cached buffer's maxsize.

See also:

Please refer to *ReplayBuffer* for other APIs' usage.

add(*batch: Batch*, *buffer_ids: Optional[Union[ndarray, List[int]]] = None*) → Tuple[ndarray, ndarray, ndarray, ndarray]

Add a batch of data into CachedReplayBuffer.

Each of the data's length (first dimension) must equal to the length of buffer_ids. By default the buffer_ids is [0, 1, ..., cached_buffer_num - 1].

Return (current_index, episode_reward, episode_length, episode_start_index) with each of the shape (len(buffer_ids), ...), where (current_index[i], episode_reward[i], episode_length[i], episode_start_index[i]) refers to the cached_buffer_ids[i]th cached buffer's corresponding episode result.

1.9.3 Collector

Collector

class tianshou.data.**Collector**(*policy: BasePolicy*, *env: Union[Env, BaseVectorEnv]*, *buffer: Optional[ReplayBuffer] = None*, *preprocess_fn: Optional[Callable[[...], Batch]] = None*, *exploration_noise: bool = False*)

Bases: `object`

Collector enables the policy to interact with different types of envs with exact number of steps or episodes.

Parameters

- **policy** – an instance of the *BasePolicy* class.
- **env** – a `gym.Env` environment or an instance of the *BaseVectorEnv* class.
- **buffer** – an instance of the *ReplayBuffer* class. If set to None, it will not store the data. Default to None.
- **preprocess_fn** (*function*) – a function called before the data has been added to the buffer, see issue #42 and *Handle Batched Data Stream in Collector*. Default to None.
- **exploration_noise** (*bool*) – determine whether the action needs to be modified with corresponding policy's exploration noise. If so, "policy. exploration_noise(act, batch)" will be called automatically to add the exploration noise into action. Default to False.

The "preprocess_fn" is a function called before the data has been added to the buffer with batch format. It will receive only "obs" and "env_id" when the collector resets the environment, and will receive the keys "obs_next", "rew", "terminated", "truncated, "info", "policy" and "env_id" in a normal env step. Alternatively, it may also accept the keys "obs_next", "rew", "done", "info", "policy" and "env_id". It returns either a dict or a *Batch* with the modified keys and values. Examples are in "test/base/test_collector.py".

Note: Please make sure the given environment has a time limitation if using n_episode collect option.

Note: In past versions of Tianshou, the replay buffer that was passed to *__init__* was automatically reset. This is not done in the current implementation.

---

reset(*reset_buffer: bool = True*, *gym_reset_kwargs: Optional[Dict[str, Any]] = None*) → None

Reset the environment, statistics, current data and possibly replay memory.

Parameters

- **reset_buffer** (*bool*) – if true, reset the replay buffer that is attached to the collector.
- **gym_reset_kwargs** – extra keyword arguments to pass into the environment's reset function. Defaults to None (extra keyword arguments)

reset_stat() → None

Reset the statistic variables.

reset_buffer(*keep_statistics: bool = False*) → None

Reset the data buffer.

reset_env(*gym_reset_kwargs: Optional[Dict[str, Any]] = None*) → None

Reset all of the environments.

collect(*n_step: Optional[int] = None*, *n_episode: Optional[int] = None*, *random: bool = False*, *render: Optional[float] = None*, *no_grad: bool = True*, *gym_reset_kwargs: Optional[Dict[str, Any]] = None*) → Dict[str, Any]

Collect a specified number of step or episode.

To ensure unbiased sampling result with n_episode option, this function will first collect `n_episode - env_num` episodes, then for the last `env_num` episodes, they will be collected evenly from each env.

Parameters

- **n_step** (*int*) – how many steps you want to collect.
- **n_episode** (*int*) – how many episodes you want to collect.
- **random** (*bool*) – whether to use random policy for collecting data. Default to False.

- **render** (*float*) – the sleep time between rendering consecutive frames. Default to None (no rendering).
- **no_grad** (*bool*) – whether to retain gradient in policy.forward(). Default to True (no gradient retaining).
- **gym_reset_kwargs** – extra keyword arguments to pass into the environment's reset function. Defaults to None (extra keyword arguments)

Note: One and only one collection number specification is permitted, either `n_step` or `n_episode`.

Returns

A dict including the following keys

- `n/ep` collected number of episodes.
- `n/st` collected number of steps.
- `rews` array of episode reward over collected episodes.
- `lens` array of episode length over collected episodes.
- `idxs` array of episode start index in buffer over collected episodes.
- `rew` mean of episodic rewards.
- `len` mean of episodic lengths.
- `rew_std` standard error of episodic rewards.
- `len_std` standard error of episodic lengths.

AsyncCollector

class tianshou.data.**AsyncCollector**(*policy: BasePolicy*, *env: BaseVectorEnv*, *buffer: Optional[ReplayBuffer] = None*, *preprocess_fn: Optional[Callable[[...], Batch]] = None*, *exploration_noise: bool = False*)

Bases: *Collector*

Async Collector handles async vector environment.

The arguments are exactly the same as *Collector*, please refer to *Collector* for more detailed explanation.

reset_env(*gym_reset_kwargs: Optional[Dict[str, Any]] = None*) → None

Reset all of the environments.

collect(*n_step: Optional[int] = None*, *n_episode: Optional[int] = None*, *random: bool = False*, *render: Optional[float] = None*, *no_grad: bool = True*, *gym_reset_kwargs: Optional[Dict[str, Any]] = None*) → Dict[str, Any]

Collect a specified number of step or episode with async env setting.

This function doesn't collect exactly n_step or n_episode number of transitions. Instead, in order to support async setting, it may collect more than given n_step or n_episode transitions and save into buffer.

Parameters

- **n_step** (*int*) – how many steps you want to collect.
- **n_episode** (*int*) – how many episodes you want to collect.

- **random** (*bool*) – whether to use random policy for collecting data. Default to False.
- **render** (*float*) – the sleep time between rendering consecutive frames. Default to None (no rendering).
- **no_grad** (*bool*) – whether to retain gradient in policy.forward(). Default to True (no gradient retaining).
- **gym_reset_kwargs** – extra keyword arguments to pass into the environment's reset function. Defaults to None (extra keyword arguments)

Note: One and only one collection number specification is permitted, either `n_step` or `n_episode`.

Returns

A dict including the following keys

- `n/ep` collected number of episodes.
- `n/st` collected number of steps.
- `rews` array of episode reward over collected episodes.
- `lens` array of episode length over collected episodes.
- `idxs` array of episode start index in buffer over collected episodes.
- `rew` mean of episodic rewards.
- `len` mean of episodic lengths.
- `rew_std` standard error of episodic rewards.
- `len_std` standard error of episodic lengths.

1.9.4 Utils

to_numpy

`tianshou.data.`**to_numpy**(*x: Any*) → Union[*Batch*, ndarray]

Return an object without torch.Tensor.

to_torch

`tianshou.data.`**to_torch**(*x: Any*, *dtype: Optional[dtype] = None*, *device: Union[str, int, device] = 'cpu'*) → Union[*Batch*, Tensor]

Return an object without np.ndarray.

to_torch_as

tianshou.data.**to_torch_as**(*x: Any*, *y: Tensor*) → Union[*Batch*, Tensor]

Return an object without np.ndarray.

Same as `to_torch(x, dtype=y.dtype, device=y.device)`.

SegmentTree

class tianshou.data.**SegmentTree**(*size: int*)

Bases: `object`

Implementation of Segment Tree.

The segment tree stores an array `arr` with size `n`. It supports value update and fast query of the sum for the interval `[left, right)` in O(log n) time. The detailed procedure is as follows:

1. Pad the array to have length of power of 2, so that leaf nodes in the segment tree have the same depth.
2. Store the segment tree in a binary heap.

Parameters
size (*int*) – the size of segment tree.

__len__() → int

__getitem__(*index: Union[int, ndarray]*) → Union[float, ndarray]

Return self[index].

__setitem__(*index: Union[int, ndarray]*, *value: Union[float, ndarray]*) → None

Update values in segment tree.

Duplicate values in `index` are handled by numpy: later index overwrites previous ones.

```
>>> a = np.array([1, 2, 3, 4])
>>> a[[0, 1, 0, 1]] = [4, 5, 6, 7]
>>> print(a)
[6 7 3 4]
```

reduce(*start: int = 0*, *end: Optional[int] = None*) → float

Return operation(value[start:end]).

get_prefix_sum_idx(*value: Union[float, ndarray]*) → Union[int, ndarray]

Find the index with given value.

Return the minimum index for each `v` in `value` so that $v \leq \text{sums}_i$, where $\text{sums}_i = \sum_{j=0}^{i} \text{arr}_j$.

> **Warning:** Please make sure all of the values inside the segment tree are non-negative when using this function.

1.10 tianshou.env

1.10.1 VectorEnv

BaseVectorEnv

class tianshou.env.**BaseVectorEnv**(*env_fns: List[Callable[[], Union[Env, Env, PettingZooEnv]]]*, *worker_fn: Callable[[Callable[[], Env]], EnvWorker]*, *wait_num: Optional[int] = None*, *timeout: Optional[float] = None*)

Bases: `object`

Base class for vectorized environments.

Usage:

```
env_num = 8
envs = DummyVectorEnv([lambda: gym.make(task) for _ in range(env_num)])
assert len(envs) == env_num
```

It accepts a list of environment generators. In other words, an environment generator `efn` of a specific task means that `efn()` returns the environment of the given task, for example, `gym.make(task)`.

All of the VectorEnv must inherit *BaseVectorEnv*. Here are some other usages:

```
envs.seed(2)  # which is equal to the next line
envs.seed([2, 3, 4, 5, 6, 7, 8, 9])  # set specific seed for each env
obs = envs.reset()  # reset all environments
obs = envs.reset([0, 5, 7])  # reset 3 specific environments
obs, rew, done, info = envs.step([1] * 8)  # step synchronously
envs.render()  # render all environments
envs.close()  # close all environments
```

Warning: If you use your own environment, please make sure the `seed` method is set up properly, e.g.,

```
def seed(self, seed):
    np.random.seed(seed)
```

Otherwise, the outputs of these envs may be the same with each other.

Parameters

- **env_fns** – a list of callable envs, `env_fns[i]()` generates the i-th env.
- **worker_fn** – a callable worker, `worker_fn(env_fns[i])` generates a worker which contains the i-th env.
- **wait_num** (*int*) – use in asynchronous simulation if the time cost of `env.step` varies with time and synchronously waiting for all environments to finish a step is time-wasting. In that case, we can return when `wait_num` environments finish a step and keep on simulation in these environments. If `None`, asynchronous simulation is disabled; else, `1 <= wait_num <= env_num`.
- **timeout** (*float*) – use in asynchronous simulation same as above, in each vectorized step it only deal with those environments spending time within `timeout` seconds.

__len__() → int

Return len(self), which is the number of environments.

get_env_attr(*key: str*, *id: Optional[Union[int, List[int], ndarray]] = None*) → List[Any]

Get an attribute from the underlying environments.

If id is an int, retrieve the attribute denoted by key from the environment underlying the worker at index id. The result is returned as a list with one element. Otherwise, retrieve the attribute for all workers at indices id and return a list that is ordered correspondingly to id.

Parameters

- **key** (*str*) – The key of the desired attribute.
- **id** – Indice(s) of the desired worker(s). Default to None for all env_id.

Return list

The list of environment attributes.

set_env_attr(*key: str*, *value: Any*, *id: Optional[Union[int, List[int], ndarray]] = None*) → None

Set an attribute in the underlying environments.

If id is an int, set the attribute denoted by key from the environment underlying the worker at index id to value. Otherwise, set the attribute for all workers at indices id.

Parameters

- **key** (*str*) – The key of the desired attribute.
- **value** (*Any*) – The new value of the attribute.
- **id** – Indice(s) of the desired worker(s). Default to None for all env_id.

reset(*id: Optional[Union[int, List[int], ndarray]] = None*, ***kwargs: Any*) → Tuple[ndarray, Union[dict, List[dict]]]

Reset the state of some envs and return initial observations.

If id is None, reset the state of all the environments and return initial observations, otherwise reset the specific environments with the given id, either an int or a list.

step(*action: ndarray*, *id: Optional[Union[int, List[int], ndarray]] = None*) → Tuple[ndarray, ndarray, ndarray, ndarray, ndarray]

Run one timestep of some environments' dynamics.

If id is None, run one timestep of all the environments' dynamics; otherwise run one timestep for some environments with given id, either an int or a list. When the end of episode is reached, you are responsible for calling reset(id) to reset this environment's state.

Accept a batch of action and return a tuple (batch_obs, batch_rew, batch_done, batch_info) in numpy format.

Parameters

action (*numpy.ndarray*) – a batch of action provided by the agent.

Returns

A tuple consisting of either:

- `obs` a numpy.ndarray, the agent's observation of current environments
- `rew` a numpy.ndarray, the amount of rewards returned after previous actions
- `terminated` a numpy.ndarray, whether these episodes have been terminated
- `truncated` a numpy.ndarray, whether these episodes have been truncated

- `info` a numpy.ndarray, contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

For the async simulation:

Provide the given action to the environments. The action sequence should correspond to the `id` argument, and the `id` argument should be a subset of the `env_id` in the last returned `info` (initially they are env_ids of all the environments). If action is None, fetch unfinished step() calls instead.

seed(*seed: Optional[Union[int, List[int]]] = None*) → List[Optional[List[int]]]

Set the seed for all environments.

Accept `None`, an int (which will extend `i` to `[i, i + 1, i + 2, ...]`) or a list.

Returns

The list of seeds used in this env's random number generators. The first value in the list should be the "main" seed, or the value which a reproducer pass to "seed".

render(***kwargs: Any*) → List[Any]

Render all of the environments.

close() → None

Close all of the environments.

This function will be called only once (if not, it will be called during garbage collected). This way, `close` of all workers can be assured.

DummyVectorEnv

class tianshou.env.**DummyVectorEnv**(*env_fns: List[Callable[[], Union[Env, Env, PettingZooEnv]]]*, ***kwargs: Any*)

Bases: *BaseVectorEnv*

Dummy vectorized environment wrapper, implemented in for-loop.

See also:

Please refer to *BaseVectorEnv* for other APIs' usage.

SubprocVectorEnv

class tianshou.env.**SubprocVectorEnv**(*env_fns: List[Callable[[], Union[Env, Env, PettingZooEnv]]]*, ***kwargs: Any*)

Bases: *BaseVectorEnv*

Vectorized environment wrapper based on subprocess.

See also:

Please refer to *BaseVectorEnv* for other APIs' usage.

ShmemVectorEnv

class tianshou.env.**ShmemVectorEnv**(*env_fns: List[Callable[[], Union[Env, Env, PettingZooEnv]]]*, ***kwargs: Any*)

Bases: *BaseVectorEnv*

Optimized SubprocVectorEnv with shared buffers to exchange observations.

ShmemVectorEnv has exactly the same API as SubprocVectorEnv.

See also:

Please refer to *BaseVectorEnv* for other APIs' usage.

RayVectorEnv

class tianshou.env.**RayVectorEnv**(*env_fns: List[Callable[[], Union[Env, Env, PettingZooEnv]]]*, ***kwargs: Any*)

Bases: *BaseVectorEnv*

Vectorized environment wrapper based on ray.

This is a choice to run distributed environments in a cluster.

See also:

Please refer to *BaseVectorEnv* for other APIs' usage.

1.10.2 Wrapper

ContinuousToDiscrete

class tianshou.env.**ContinuousToDiscrete**(*env: Env*, *action_per_dim: Union[int, List[int]]*)

Bases: `ActionWrapper`

Gym environment wrapper to take discrete action in a continuous environment.

Parameters

- **env** (*gym.Env*) – gym environment with continuous action space.
- **action_per_dim** (*int*) – number of discrete actions in each dimension of the action space.

action(*act: ndarray*) → ndarray

Returns a modified action before `env.step()` is called.

Args:
action: The original `step()` actions

Returns:
The modified actions

VectorEnvWrapper

class tianshou.env.**VectorEnvWrapper**(*venv:* BaseVectorEnv)

Bases: *BaseVectorEnv*

Base class for vectorized environments wrapper.

__len__() → int

Return len(self), which is the number of environments.

get_env_attr(*key: str*, *id: Optional[Union[int, List[int], ndarray]] = None*) → List[Any]

Get an attribute from the underlying environments.

If id is an int, retrieve the attribute denoted by key from the environment underlying the worker at index id. The result is returned as a list with one element. Otherwise, retrieve the attribute for all workers at indices id and return a list that is ordered correspondingly to id.

Parameters

- **key** (*str*) – The key of the desired attribute.
- **id** – Indice(s) of the desired worker(s). Default to None for all env_id.

Return list
The list of environment attributes.

set_env_attr(*key: str*, *value: Any*, *id: Optional[Union[int, List[int], ndarray]] = None*) → None

Set an attribute in the underlying environments.

If id is an int, set the attribute denoted by key from the environment underlying the worker at index id to value. Otherwise, set the attribute for all workers at indices id.

Parameters

- **key** (*str*) – The key of the desired attribute.
- **value** (*Any*) – The new value of the attribute.
- **id** – Indice(s) of the desired worker(s). Default to None for all env_id.

reset(*id: Optional[Union[int, List[int], ndarray]] = None*, ***kwargs: Any*) → Tuple[ndarray, Union[dict, List[dict]]]

Reset the state of some envs and return initial observations.

If id is None, reset the state of all the environments and return initial observations, otherwise reset the specific environments with the given id, either an int or a list.

step(*action: ndarray*, *id: Optional[Union[int, List[int], ndarray]] = None*) → Tuple[ndarray, ndarray, ndarray, ndarray, ndarray]

Run one timestep of some environments' dynamics.

If id is None, run one timestep of all the environments' dynamics; otherwise run one timestep for some environments with given id, either an int or a list. When the end of episode is reached, you are responsible for calling reset(id) to reset this environment's state.

Accept a batch of action and return a tuple (batch_obs, batch_rew, batch_done, batch_info) in numpy format.

Parameters
action (*numpy.ndarray*) – a batch of action provided by the agent.

Returns

A tuple consisting of either:

- `obs` a numpy.ndarray, the agent's observation of current environments
- `rew` a numpy.ndarray, the amount of rewards returned after previous actions
- `terminated` a numpy.ndarray, whether these episodes have been terminated
- `truncated` a numpy.ndarray, whether these episodes have been truncated
- `info` a numpy.ndarray, contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

For the async simulation:

Provide the given action to the environments. The action sequence should correspond to the `id` argument, and the `id` argument should be a subset of the `env_id` in the last returned `info` (initially they are env_ids of all the environments). If action is None, fetch unfinished step() calls instead.

seed(*seed: Optional[Union[int, List[int]]] = None*) → List[Optional[List[int]]]

Set the seed for all environments.

Accept `None`, an int (which will extend `i` to `[i, i + 1, i + 2, ...]`) or a list.

Returns
The list of seeds used in this env's random number generators. The first value in the list should be the "main" seed, or the value which a reproducer pass to "seed".

render(***kwargs: Any*) → List[Any]

Render all of the environments.

close() → None

Close all of the environments.

This function will be called only once (if not, it will be called during garbage collected). This way, `close` of all workers can be assured.

VectorEnvNormObs

class tianshou.env.**VectorEnvNormObs**(*venv:* BaseVectorEnv, *update_obs_rms: bool = True*)

Bases: *VectorEnvWrapper*

An observation normalization wrapper for vectorized environments.

Parameters
update_obs_rms (*bool*) – whether to update obs_rms. Default to True.

reset(*id: Optional[Union[int, List[int], ndarray]] = None*, ***kwargs: Any*) → Tuple[ndarray, Union[dict, List[dict]]]

Reset the state of some envs and return initial observations.

If id is None, reset the state of all the environments and return initial observations, otherwise reset the specific environments with the given id, either an int or a list.

step(*action: ndarray*, *id: Optional[Union[int, List[int], ndarray]] = None*) → Tuple[ndarray, ndarray, ndarray, ndarray, ndarray]

Run one timestep of some environments' dynamics.

If id is None, run one timestep of all the environments' dynamics; otherwise run one timestep for some environments with given id, either an int or a list. When the end of episode is reached, you are responsible for calling reset(id) to reset this environment's state.

Accept a batch of action and return a tuple (batch_obs, batch_rew, batch_done, batch_info) in numpy format.

Parameters
action (*numpy.ndarray*) – a batch of action provided by the agent.

Returns

A tuple consisting of either:

- `obs` a numpy.ndarray, the agent's observation of current environments
- `rew` a numpy.ndarray, the amount of rewards returned after previous actions
- `terminated` a numpy.ndarray, whether these episodes have been terminated
- `truncated` a numpy.ndarray, whether these episodes have been truncated
- `info` a numpy.ndarray, contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

For the async simulation:

Provide the given action to the environments. The action sequence should correspond to the `id` argument, and the `id` argument should be a subset of the `env_id` in the last returned `info` (initially they are env_ids of all the environments). If action is None, fetch unfinished step() calls instead.

set_obs_rms(*obs_rms:* RunningMeanStd) → None

Set with given observation running mean/std.

get_obs_rms() → *RunningMeanStd*

Return observation running mean/std.

1.10.3 Worker

EnvWorker

class tianshou.env.worker.**EnvWorker**(*env_fn: Callable[[], Env]*)

Bases: ABC

An abstract worker for an environment.

abstract get_env_attr(*key: str*) → Any

abstract set_env_attr(*key: str*, *value: Any*) → None

send(*action: Optional[ndarray]*) → None

Send action signal to low-level worker.

When action is None, it indicates sending "reset" signal; otherwise it indicates "step" signal. The paired return value from "recv" function is determined by such kind of different signal.

recv() → Union[Tuple[ndarray, ndarray, ndarray, ndarray, ndarray], Tuple[ndarray, dict]]

Receive result from low-level worker.

If the last "send" function sends a NULL action, it only returns a single observation; otherwise it returns a tuple of (obs, rew, done, info) or (obs, rew, terminated, truncated, info), based on whether the environment is using the old step API or the new one.

abstract reset(***kwargs: Any*) → Tuple[ndarray, dict]

step(*action: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray, ndarray]

Perform one timestep of the environment's dynamic.

"send" and "recv" are coupled in sync simulation, so users only call "step" function. But they can be called separately in async simulation, i.e. someone calls "send" first, and calls "recv" later.

static wait(*workers: List[*EnvWorker*]*, *wait_num: int*, *timeout: Optional[float] = None*) → List[*EnvWorker*]

Given a list of workers, return those ready ones.

seed(*seed: Optional[int] = None*) → Optional[List[int]]

abstract render(***kwargs: Any*) → Any

Render the environment.

abstract close_env() → None

close() → None

DummyEnvWorker

class tianshou.env.worker.**DummyEnvWorker**(*env_fn: Callable[[], Env]*)

Bases: *EnvWorker*

Dummy worker used in sequential vector environments.

get_env_attr(*key: str*) → Any

set_env_attr(*key: str*, *value: Any*) → None

reset(***kwargs: Any*) → Tuple[ndarray, dict]

static wait(*workers: List[*DummyEnvWorker*]*, *wait_num: int*, *timeout: Optional[float] = None*) → List[*DummyEnvWorker*]

Given a list of workers, return those ready ones.

send(*action: Optional[ndarray]*, ***kwargs: Any*) → None

Send action signal to low-level worker.

When action is None, it indicates sending "reset" signal; otherwise it indicates "step" signal. The paired return value from "recv" function is determined by such kind of different signal.

seed(*seed: Optional[int] = None*) → Optional[List[int]]

render(***kwargs: Any*) → Any

Render the environment.

close_env() → None

SubprocEnvWorker

class tianshou.env.worker.**SubprocEnvWorker**(*env_fn: Callable[[], Env]*, *share_memory: bool = False*)

Bases: *EnvWorker*

Subprocess worker used in SubprocVectorEnv and ShmemVectorEnv.

get_env_attr(*key: str*) → Any

set_env_attr(*key: str*, *value: Any*) → None

static wait(*workers: List[*SubprocEnvWorker*]*, *wait_num: int*, *timeout: Optional[float] = None*) → List[*SubprocEnvWorker*]

Given a list of workers, return those ready ones.

send(*action: Optional[ndarray]*, ***kwargs: Any*) → None

Send action signal to low-level worker.

When action is None, it indicates sending "reset" signal; otherwise it indicates "step" signal. The paired return value from "recv" function is determined by such kind of different signal.

recv() → Union[Tuple[ndarray, ndarray, ndarray, ndarray, ndarray], Tuple[ndarray, dict]]

Receive result from low-level worker.

If the last "send" function sends a NULL action, it only returns a single observation; otherwise it returns a tuple of (obs, rew, done, info) or (obs, rew, terminated, truncated, info), based on whether the environment is using the old step API or the new one.

reset(***kwargs: Any*) → Tuple[ndarray, dict]

seed(*seed: Optional[int] = None*) → Optional[List[int]]

render(***kwargs: Any*) → Any

Render the environment.

close_env() → None

RayEnvWorker

class tianshou.env.worker.**RayEnvWorker**(*env_fn: Callable[[], Env]*)

Bases: *EnvWorker*

Ray worker used in RayVectorEnv.

get_env_attr(*key: str*) → Any

set_env_attr(*key: str*, *value: Any*) → None

reset(***kwargs: Any*) → Any

static wait(*workers: List[*RayEnvWorker*]*, *wait_num: int*, *timeout: Optional[float] = None*) → List[*RayEnvWorker*]

Given a list of workers, return those ready ones.

send(*action: Optional[ndarray]*, ***kwargs: Any*) → None

Send action signal to low-level worker.

When action is None, it indicates sending "reset" signal; otherwise it indicates "step" signal. The paired return value from "recv" function is determined by such kind of different signal.

recv() → Tuple[ndarray, ndarray, ndarray, ndarray, ndarray]

Receive result from low-level worker.

If the last "send" function sends a NULL action, it only returns a single observation; otherwise it returns a tuple of (obs, rew, done, info) or (obs, rew, terminated, truncated, info), based on whether the environment is using the old step API or the new one.

seed(*seed: Optional[int] = None*) → Optional[List[int]]

render(***kwargs: Any*) → Any

Render the environment.

close_env() → None

1.10.4 Utils

PettingZooEnv

class tianshou.env.**PettingZooEnv**(*env: BaseWrapper*)

Bases: AECEnv, ABC

The interface for petting zoo environments.

Multi-agent environments must be wrapped as *PettingZooEnv*. Here is the usage:

```
env = PettingZooEnv(...)
# obs is a dict containing obs, agent_id, and mask
obs = env.reset()
action = policy(obs)
obs, rew, trunc, term, info = env.step(action)
env.close()
```

The available action's mask is set to True, otherwise it is set to False. Further usage can be found at *Multi-Agent Reinforcement Learning*.

agents: List[AgentID]

rewards: Dict[AgentID, float]

reset(**args: Any*, ***kwargs: Any*) → Tuple[dict, dict]

Resets the environment to a starting state.

step(*action: Any*) → Tuple[Dict, List[int], bool, bool, Dict]

Accepts and executes the action of the current agent_selection in the environment.

Automatically switches control to the next agent.

close() → None

Closes any resources that should be released.

Closes the rendering window, subprocesses, network connections, or any other resources that should be released.

seed(*seed: Optional[Any] = None*) → None

Reseeds the environment (making the resulting environment deterministic).

render() → Any

Renders the environment as specified by self.render_mode.

Render mode can be *human* to display a window. Other render modes in the default environments are *'rgb_array'* which returns a numpy array and is supported by all environments outside of classic, and *'ansi'* which returns the strings printed (specific to classic environments).

metadata: Dict[str, Any]

possible_agents: List[AgentID]

observation_spaces: Dict[AgentID, gymnasium.spaces.Space]

action_spaces: Dict[AgentID, gymnasium.spaces.Space]

terminations: Dict[AgentID, bool]

truncations: Dict[AgentID, bool]

infos: Dict[AgentID, Dict[str, Any]]

agent_selection: AgentID

1.11 tianshou.policy

1.11.1 Base

class tianshou.policy.**BasePolicy**(*observation_space: Optional[Space] = None*, *action_space: Optional[Space] = None*, *action_scaling: bool = False*, *action_bound_method: str = ''*, *lr_scheduler: Optional[Union[LambdaLR, MultipleLRSchedulers]] = None*)

Bases: ABC, Module

The base class for any RL policy.

Tianshou aims to modularize RL algorithms. It comes into several classes of policies in Tianshou. All of the policy classes must inherit *BasePolicy*.

A policy class typically has the following parts:

- __init__(): initialize the policy, including coping the target network and so on;
- *forward()*: compute action with given observation;
- *process_fn()*: pre-process data from the replay buffer (this function can interact with replay buffer);
- *learn()*: update policy with a given batch of data.
- *post_process_fn()*: update the replay buffer from the learning process (e.g., prioritized replay buffer needs to update the weight);
- *update()*: the main interface for training, i.e., *process_fn* -> *learn* -> *post_process_fn*.

Most of the policy needs a neural network to predict the action and an optimizer to optimize the policy. The rules of self-defined networks are:

1. Input: observation "obs" (may be a numpy.ndarray, a torch.Tensor, a dict or any others), hidden state "state" (for RNN usage), and other information "info" provided by the environment.

2. Output: some "logits", the next hidden state "state", and the intermediate result during policy forwarding procedure "policy". The "logits" could be a tuple instead of a `torch.Tensor`. It depends on how the policy process the network output. For example, in PPO, the return of the network might be `(mu, sigma), state` for Gaussian policy. The "policy" can be a Batch of torch.Tensor or other things, which will be stored in the replay buffer, and can be accessed in the policy update process (e.g. in "policy.learn()", the "batch.policy" is what you need).

Since *BasePolicy* inherits `torch.nn.Module`, you can use *BasePolicy* almost the same as `torch.nn.Module`, for instance, loading and saving the model:

```
torch.save(policy.state_dict(), "policy.pth")
policy.load_state_dict(torch.load("policy.pth"))
```

set_agent_id(*agent_id: int*) → None

Set self.agent_id = agent_id, for MARL.

exploration_noise(*act: Union[ndarray,* Batch*]*, *batch:* Batch) → Union[ndarray, *Batch*]

Modify the action from policy.forward with exploration noise.

Parameters

- **act** – a data batch or numpy.ndarray which is the action taken by policy.forward.
- **batch** – the input batch for policy.forward, kept for advanced usage.

Returns

action in the same form of input "act" but with added exploration noise.

soft_update(*tgt: Module*, *src: Module*, *tau: float*) → None

Softly update the parameters of target module towards the parameters of source module.

abstract forward(*batch:* Batch, *state: Optional[Union[dict,* Batch*, ndarray]] = None*, ***kwargs: Any*) → *Batch*

Compute action over the given batch data.

Returns

A *Batch* which MUST have the following keys:

- `act` an numpy.ndarray or a torch.Tensor, the action over given batch data.
- `state` a dict, an numpy.ndarray or a torch.Tensor, the internal state of the policy, `None` as default.

Other keys are user-defined. It depends on the algorithm. For example,

```
# some code
return Batch(logits=..., act=..., state=None, dist=...)
```

The keyword `policy` is reserved and the corresponding data will be stored into the replay buffer. For instance,

```
# some code
return Batch(..., policy=Batch(log_prob=dist.log_prob(act)))
# and in the sampled data batch, you can directly use
# batch.policy.log_prob to get your data.
```

Note: In continuous action space, you should do another step "map_action" to get the real action:

```
act = policy(batch).act  # doesn't map to the target action range
act = policy.map_action(act, batch)
```

map_action(*act: Union[*Batch*, ndarray]*) → Union[*Batch*, ndarray]

Map raw network output to action range in gym's env.action_space.

This function is called in *collect()* and only affects action sending to env. Remapped action will not be stored in buffer and thus can be viewed as a part of env (a black box action transformation).

Action mapping includes 2 standard procedures: bounding and scaling. Bounding procedure expects original action range is (-inf, inf) and maps it to [-1, 1], while scaling procedure expects original action range is (-1, 1) and maps it to [action_space.low, action_space.high]. Bounding procedure is applied first.

Parameters

act – a data batch or numpy.ndarray which is the action taken by policy.forward.

Returns

action in the same form of input "act" but remap to the target action space.

map_action_inverse(*act: Union[*Batch*, List, ndarray]*) → Union[*Batch*, List, ndarray]

Inverse operation to *map_action()*.

This function is called in *collect()* for random initial steps. It scales [action_space.low, action_space.high] to the value ranges of policy.forward.

Parameters

act – a data batch, list or numpy.ndarray which is the action taken by gym.spaces.Box.sample().

Returns

action remapped.

process_fn(*batch:* Batch, *buffer:* ReplayBuffer, *indices: ndarray*) → *Batch*

Pre-process the data from the provided replay buffer.

Used in *update()*. Check out *policy.process_fn* for more information.

abstract learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, Any]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

post_process_fn(*batch:* Batch, *buffer:* ReplayBuffer, *indices: ndarray*) → None

Post-process the data from the provided replay buffer.

Typical usage is to update the sampling weight in prioritized experience replay. Used in *update()*.

update(*sample_size: int*, *buffer: Optional[*ReplayBuffer*]*, ***kwargs: Any*) → Dict[str, Any]

Update the policy network and replay buffer.

It includes 3 function steps: process_fn, learn, and post_process_fn. In addition, this function will change the value of `self.updating`: it will be False before this function and will be True when executing *update()*. Please refer to *States for policy* for more detailed explanation.

Parameters

- **sample_size** (*int*) – 0 means it will extract all the data from the buffer, otherwise it will sample a batch with given sample_size.
- **buffer** (ReplayBuffer) – the corresponding replay buffer.

Returns

A dict, including the data needed to be logged (e.g., loss) from `policy.learn()`.

static value_mask(*buffer:* ReplayBuffer, *indices: ndarray*) → ndarray

Value mask determines whether the obs_next of buffer[indices] is valid.

For instance, usually "obs_next" after "done" flag is considered to be invalid, and its q/advantage value can provide meaningless (even misleading) information, and should be set to 0 by hand. But if "done" flag is generated because timelimit of game length (info["TimeLimit.truncated"] is set to True in gym's settings), "obs_next" will instead be valid. Value mask is typically used for assisting in calculating the correct q/advantage value.

Parameters

- **buffer** (ReplayBuffer) – the corresponding replay buffer.
- **indices** (*numpy.ndarray*) – indices of replay buffer whose "obs_next" will be judged.

Returns

A bool type numpy.ndarray in the same shape with indices. "True" means "obs_next" of that buffer[indices] is valid.

static compute_episodic_return(*batch:* Batch, *buffer:* ReplayBuffer, *indices: ndarray*, *v_s_: Optional[Union[ndarray, Tensor]] = None*, *v_s: Optional[Union[ndarray, Tensor]] = None*, *gamma: float = 0.99*, *gae_lambda: float = 0.95*) → Tuple[ndarray, ndarray]

Compute returns over given batch.

Use Implementation of Generalized Advantage Estimator (arXiv:1506.02438) to calculate q/advantage value of given batch.

Parameters

- **batch** (Batch) – a data batch which contains several episodes of data in sequential order. Mind that the end of each finished episode of batch should be marked by done flag, unfinished (or collecting) episodes will be recognized by buffer.unfinished_index().
- **indices** (*numpy.ndarray*) – tell batch's location in buffer, batch is equal to buffer[indices].
- **v_s** (*np.ndarray*) – the value function of all next states $V(s')$.
- **gamma** (*float*) – the discount factor, should be in [0, 1]. Default to 0.99.

- **gae_lambda** (*float*) – the parameter for Generalized Advantage Estimation, should be in [0, 1]. Default to 0.95.

Returns

two numpy arrays (returns, advantage) with each shape (bsz,).

static compute_nstep_return(*batch: Batch*, *buffer: ReplayBuffer*, *indice: ndarray*, *target_q_fn: Callable[[ReplayBuffer, ndarray], Tensor]*, *gamma: float = 0.99*, *n_step: int = 1*, *rew_norm: bool = False*) → *Batch*

Compute n-step return for Q-learning targets.

$$G_t = \sum_{i=t}^{t+n-1} \gamma^{i-t}(1-d_i)r_i + \gamma^n(1-d_{t+n})Q_{\text{target}}(s_{t+n})$$

where γ is the discount factor, $\gamma \in [0, 1]$, d_t is the done flag of step t.

Parameters

- **batch** (Batch) – a data batch, which is equal to buffer[indice].
- **buffer** (ReplayBuffer) – the data buffer.
- **target_q_fn** (*function*) – a function which compute target Q value of "obs_next" given data buffer and wanted indices.
- **gamma** (*float*) – the discount factor, should be in [0, 1]. Default to 0.99.
- **n_step** (*int*) – the number of estimation step, should be an int greater than 0. Default to 1.
- **rew_norm** (*bool*) – normalize the reward to Normal(0, 1), Default to False.

Returns

a Batch. The result will be stored in batch.returns as a torch.Tensor with the same shape as target_q_fn's return tensor.

training: bool

class tianshou.policy.**RandomPolicy**(*observation_space: Optional[Space] = None*, *action_space: Optional[Space] = None*, *action_scaling: bool = False*, *action_bound_method: str = ''*, *lr_scheduler: Optional[Union[LambdaLR, MultipleLRSchedulers]] = None*)

Bases: *BasePolicy*

A random agent used in multi-agent learning.

It randomly chooses an action from the legal action.

forward(*batch: Batch*, *state: Optional[Union[dict, Batch, ndarray]] = None*, ***kwargs: Any*) → *Batch*

Compute the random action over the given batch data.

The input should contain a mask in batch.obs, with "True" to be available and "False" to be unavailable. For example, `batch.obs.mask == np.array([[False, True, False]])` means with batch size 1, action "1" is available but action "0" and "2" are unavailable.

Returns

A *Batch* with "act" key, containing the random action.

See also:

Please refer to *forward()* for more detailed explanation.

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Since a random agent learns nothing, it returns an empty dict.

training: bool

1.11.2 Model-free

DQN Family

class tianshou.policy.**DQNPolicy**(*model: Module*, *optim: Optimizer*, *discount_factor: float = 0.99*, *estimation_step: int = 1*, *target_update_freq: int = 0*, *reward_normalization: bool = False*, *is_double: bool = True*, *clip_loss_grad: bool = False*, ***kwargs: Any*)

Bases: *BasePolicy*

Implementation of Deep Q Network. arXiv:1312.5602.

Implementation of Double Q-Learning. arXiv:1509.06461.

Implementation of Dueling DQN. arXiv:1511.06581 (the dueling DQN is implemented in the network side, not here).

Parameters

- **model** (*torch.nn.Module*) – a model following the rules in *BasePolicy*. (s -> logits)
- **optim** (*torch.optim.Optimizer*) – a torch.optim for optimizing the model.
- **discount_factor** (*float*) – in [0, 1].
- **estimation_step** (*int*) – the number of steps to look ahead. Default to 1.
- **target_update_freq** (*int*) – the target network update frequency (0 if you do not use the target network). Default to 0.
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1). Default to False.
- **is_double** (*bool*) – use double dqn. Default to True.
- **clip_loss_grad** (*bool*) – clip the gradient of the loss in accordance with nature14236; this amounts to using the Huber loss instead of the MSE loss. Default to False.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *BasePolicy* for more detailed explanation.

set_eps(*eps: float*) → None

Set the eps for epsilon-greedy exploration.

train(*mode: bool = True*) → *DQNPolicy*

Set the module in training mode, except for the target network.

sync_weight() → None

Synchronize the weight for the target network.

process_fn(*batch: Batch*, *buffer: ReplayBuffer*, *indices: ndarray*) → *Batch*

Compute the n-step return for Q-learning targets.

More details can be found at *compute_nstep_return()*.

compute_q_value(*logits: Tensor*, *mask: Optional[ndarray]*) → Tensor

Compute the q value based on the network's raw output and action mask.

forward(*batch: Batch*, *state: Optional[Union[dict, Batch, ndarray]] = None*, *model: str = 'model'*, *input: str = 'obs'*, ***kwargs: Any*) → *Batch*

Compute action over the given batch data.

If you need to mask the action, please add a "mask" into batch.obs, for example, if we have an environment that has "0/1/2" three actions:

```
batch == Batch(
    obs=Batch(
        obs="original obs, with batch_size=1 for demonstration",
        mask=np.array([[False, True, False]]),
        # action 1 is available
        # action 0 and 2 are unavailable
    ),
    ...
)
```

Returns

A *Batch* which has 3 keys:

- `act` the action.
- `logits` the network's raw output.
- `state` the hidden state.

See also:

Please refer to *forward()* for more detailed explanation.

learn(*batch: Batch*, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

exploration_noise(*act: Union[ndarray,* Batch*]*, *batch:* Batch) → Union[ndarray, *Batch*]

Modify the action from policy.forward with exploration noise.

Parameters

- **act** – a data batch or numpy.ndarray which is the action taken by policy.forward.
- **batch** – the input batch for policy.forward, kept for advanced usage.

Returns

action in the same form of input "act" but with added exploration noise.

training: bool

class tianshou.policy.**BranchingDQNPolicy**(*model:* BranchingNet, *optim: Optimizer*, *discount_factor: float = 0.99*, *estimation_step: int = 1*, *target_update_freq: int = 0*, *reward_normalization: bool = False*, *is_double: bool = True*, ***kwargs: Any*)

Bases: *DQNPolicy*

Implementation of the Branching dual Q network arXiv:1711.08946.

Parameters

- **model** (*torch.nn.Module*) – a model following the rules in *BasePolicy*. (s -> logits)
- **optim** (*torch.optim.Optimizer*) – a torch.optim for optimizing the model.
- **discount_factor** (*float*) – in [0, 1].
- **estimation_step** (*int*) – the number of steps to look ahead. Default to 1.
- **target_update_freq** (*int*) – the target network update frequency (0 if you do not use the target network). Default to 0.
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1). Default to False.
- **is_double** (*bool*) – use double network. Default to True.

See also:

Please refer to *BasePolicy* for more detailed explanation.

process_fn(*batch:* Batch, *buffer:* ReplayBuffer, *indices: ndarray*) → *Batch*

Compute the 1-step return for BDQ targets.

forward(*batch:* Batch, *state: Optional[Union[Dict,* Batch*, ndarray]] = None*, *model: str = 'model'*, *input: str = 'obs'*, ***kwargs: Any*) → *Batch*

Compute action over the given batch data.

If you need to mask the action, please add a "mask" into batch.obs, for example, if we have an environment that has "0/1/2" three actions:

```
batch == Batch(
    obs=Batch(
        obs="original obs, with batch_size=1 for demonstration",
        mask=np.array([[False, True, False]]),
        # action 1 is available
        # action 0 and 2 are unavailable
    ),
    ...
)
```

Returns

A *Batch* which has 3 keys:

- `act` the action.
- `logits` the network's raw output.
- `state` the hidden state.

See also:

Please refer to *forward()* for more detailed explanation.

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

exploration_noise(*act: Union[ndarray,* Batch*]*, *batch:* Batch) → Union[ndarray, *Batch*]

Modify the action from policy.forward with exploration noise.

Parameters

- **act** – a data batch or numpy.ndarray which is the action taken by policy.forward.
- **batch** – the input batch for policy.forward, kept for advanced usage.

Returns

action in the same form of input "act" but with added exploration noise.

training: bool

class tianshou.policy.**C51Policy**(*model: Module*, *optim: Optimizer*, *discount_factor: float = 0.99*, *num_atoms: int = 51*, *v_min: float = -10.0*, *v_max: float = 10.0*, *estimation_step: int = 1*, *target_update_freq: int = 0*, *reward_normalization: bool = False*, ***kwargs: Any*)

Bases: *DQNPolicy*

Implementation of Categorical Deep Q-Network. arXiv:1707.06887.

Parameters

- **model** (*torch.nn.Module*) – a model following the rules in *BasePolicy*. (s -> logits)
- **optim** (*torch.optim.Optimizer*) – a torch.optim for optimizing the model.
- **discount_factor** (*float*) – in [0, 1].

- **num_atoms** (*int*) – the number of atoms in the support set of the value distribution. Default to 51.
- **v_min** (*float*) – the value of the smallest atom in the support set. Default to -10.0.
- **v_max** (*float*) – the value of the largest atom in the support set. Default to 10.0.
- **estimation_step** (*int*) – the number of steps to look ahead. Default to 1.
- **target_update_freq** (*int*) – the target network update frequency (0 if you do not use the target network). Default to 0.
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1). Default to False.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *DQNPolicy* for more detailed explanation.

compute_q_value(*logits: Tensor*, *mask: Optional[ndarray]*) → Tensor

Compute the q value based on the network's raw output and action mask.

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

class tianshou.policy.**RainbowPolicy**(*model: Module*, *optim: Optimizer*, *discount_factor: float = 0.99*, *num_atoms: int = 51*, *v_min: float = -10.0*, *v_max: float = 10.0*, *estimation_step: int = 1*, *target_update_freq: int = 0*, *reward_normalization: bool = False*, ***kwargs: Any*)

Bases: *C51Policy*

Implementation of Rainbow DQN. arXiv:1710.02298.

Parameters

- **model** (*torch.nn.Module*) – a model following the rules in *BasePolicy*. (s -> logits)
- **optim** (*torch.optim.Optimizer*) – a torch.optim for optimizing the model.
- **discount_factor** (*float*) – in [0, 1].
- **num_atoms** (*int*) – the number of atoms in the support set of the value distribution. Default to 51.

- **v_min** (*float*) – the value of the smallest atom in the support set. Default to -10.0.
- **v_max** (*float*) – the value of the largest atom in the support set. Default to 10.0.
- **estimation_step** (*int*) – the number of steps to look ahead. Default to 1.
- **target_update_freq** (*int*) – the target network update frequency (0 if you do not use the target network). Default to 0.
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1). Default to False.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *C51Policy* for more detailed explanation.

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

class tianshou.policy.**QRDQNPolicy**(*model: Module*, *optim: Optimizer*, *discount_factor: float = 0.99*, *num_quantiles: int = 200*, *estimation_step: int = 1*, *target_update_freq: int = 0*, *reward_normalization: bool = False*, ***kwargs: Any*)

Bases: *DQNPolicy*

Implementation of Quantile Regression Deep Q-Network. arXiv:1710.10044.

Parameters

- **model** (*torch.nn.Module*) – a model following the rules in *BasePolicy*. (s -> logits)
- **optim** (*torch.optim.Optimizer*) – a torch.optim for optimizing the model.
- **discount_factor** (*float*) – in [0, 1].
- **num_quantiles** (*int*) – the number of quantile midpoints in the inverse cumulative distribution function of the value. Default to 200.
- **estimation_step** (*int*) – the number of steps to look ahead. Default to 1.
- **target_update_freq** (*int*) – the target network update frequency (0 if you do not use the target network).
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1). Default to False.

- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *DQNPolicy* for more detailed explanation.

compute_q_value(*logits: Tensor*, *mask: Optional[ndarray]*) → Tensor

Compute the q value based on the network's raw output and action mask.

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

class tianshou.policy.**IQNPolicy**(*model: Module*, *optim: Optimizer*, *discount_factor: float = 0.99*, *sample_size: int = 32*, *online_sample_size: int = 8*, *target_sample_size: int = 8*, *estimation_step: int = 1*, *target_update_freq: int = 0*, *reward_normalization: bool = False*, ***kwargs: Any*)

Bases: *QRDQNPolicy*

Implementation of Implicit Quantile Network. arXiv:1806.06923.

Parameters

- **model** (*torch.nn.Module*) – a model following the rules in *BasePolicy*. (s -> logits)
- **optim** (*torch.optim.Optimizer*) – a torch.optim for optimizing the model.
- **discount_factor** (*float*) – in [0, 1].
- **sample_size** (*int*) – the number of samples for policy evaluation. Default to 32.
- **online_sample_size** (*int*) – the number of samples for online model in training. Default to 8.
- **target_sample_size** (*int*) – the number of samples for target model in training. Default to 8.
- **estimation_step** (*int*) – the number of steps to look ahead. Default to 1.
- **target_update_freq** (*int*) – the target network update frequency (0 if you do not use the target network).
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1). Default to False.

- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *QRDQNPolicy* for more detailed explanation.

forward(*batch: Batch*, *state: Optional[Union[dict, Batch, ndarray]] = None*, *model: str = 'model'*, *input: str = 'obs'*, ***kwargs: Any*) → *Batch*

Compute action over the given batch data.

If you need to mask the action, please add a "mask" into batch.obs, for example, if we have an environment that has "0/1/2" three actions:

```
batch == Batch(
    obs=Batch(
        obs="original obs, with batch_size=1 for demonstration",
        mask=np.array([[False, True, False]]),
        # action 1 is available
        # action 0 and 2 are unavailable
    ),
    ...
)
```

Returns

A *Batch* which has 3 keys:

- `act` the action.
- `logits` the network's raw output.
- `state` the hidden state.

See also:

Please refer to *forward()* for more detailed explanation.

learn(*batch: Batch*, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

class tianshou.policy.**FQFPolicy**(*model: FullQuantileFunction*, *optim: Optimizer*, *fraction_model: FractionProposalNetwork*, *fraction_optim: Optimizer*, *discount_factor: float = 0.99*, *num_fractions: int = 32*, *ent_coef: float = 0.0*, *estimation_step: int = 1*, *target_update_freq: int = 0*, *reward_normalization: bool = False*, ***kwargs: Any*)

Bases: *QRDQNPolicy*

Implementation of Fully-parameterized Quantile Function. arXiv:1911.02140.

Parameters

- **model** (*torch.nn.Module*) – a model following the rules in *BasePolicy*. (s -> logits)
- **optim** (*torch.optim.Optimizer*) – a torch.optim for optimizing the model.
- **fraction_model** (FractionProposalNetwork) – a FractionProposalNetwork for proposing fractions/quantiles given state.
- **fraction_optim** (*torch.optim.Optimizer*) – a torch.optim for optimizing the fraction model above.
- **discount_factor** (*float*) – in [0, 1].
- **num_fractions** (*int*) – the number of fractions to use. Default to 32.
- **ent_coef** (*float*) – the coefficient for entropy loss. Default to 0.
- **estimation_step** (*int*) – the number of steps to look ahead. Default to 1.
- **target_update_freq** (*int*) – the target network update frequency (0 if you do not use the target network).
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1). Default to False.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *QRDQNPolicy* for more detailed explanation.

forward(*batch: Batch*, *state: Optional[Union[dict, Batch, ndarray]] = None*, *model: str = 'model'*, *input: str = 'obs'*, *fractions: Optional[Batch] = None*, ***kwargs: Any*) → *Batch*

Compute action over the given batch data.

If you need to mask the action, please add a "mask" into batch.obs, for example, if we have an environment that has "0/1/2" three actions:

```
batch == Batch(
    obs=Batch(
        obs="original obs, with batch_size=1 for demonstration",
        mask=np.array([[False, True, False]]),
        # action 1 is available
        # action 0 and 2 are unavailable
    ),
    ...
)
```

Returns

A *Batch* which has 3 keys:

- `act` the action.
- `logits` the network's raw output.
- `state` the hidden state.

See also:

Please refer to *forward()* for more detailed explanation.

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

On-policy

class tianshou.policy.**PGPolicy**(*model: Module*, *optim: Optimizer*, *dist_fn: Type[Distribution]*, *discount_factor: float = 0.99*, *reward_normalization: bool = False*, *action_scaling: bool = True*, *action_bound_method: str = 'clip'*, *deterministic_eval: bool = False*, ***kwargs: Any*)

Bases: *BasePolicy*

Implementation of REINFORCE algorithm.

Parameters

- **model** (*torch.nn.Module*) – a model following the rules in *BasePolicy*. (s -> logits)
- **optim** (*torch.optim.Optimizer*) – a torch.optim for optimizing the model.
- **dist_fn** (*Type[torch.distributions.Distribution]*) – distribution class for computing the action.
- **discount_factor** (*float*) – in [0, 1]. Default to 0.99.
- **action_scaling** (*bool*) – whether to map actions from range [-1, 1] to range [action_spaces.low, action_spaces.high]. Default to True.
- **action_bound_method** (*str*) – method to bound action to range [-1, 1], can be either "clip" (for simply clipping the action), "tanh" (for applying tanh squashing) for now, or empty string for no bounding. Default to "clip".
- **action_space** (*Optional[gym.Space]*) – env's action space, mandatory if you want to use option "action_scaling" or "action_bound_method". Default to None.

- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).
- **deterministic_eval** (*bool*) – whether to use deterministic action instead of stochastic action sampled by the policy. Default to False.

See also:

Please refer to *BasePolicy* for more detailed explanation.

process_fn(*batch: Batch*, *buffer: ReplayBuffer*, *indices: ndarray*) → *Batch*

Compute the discounted returns for each transition.

$$G_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$$

where T is the terminal time step, γ is the discount factor, $\gamma \in [0, 1]$.

forward(*batch: Batch*, *state: Optional[Union[dict, Batch, ndarray]] = None*, ***kwargs: Any*) → *Batch*

Compute action over the given batch data.

Returns

A *Batch* which has 4 keys:

- `act` the action.
- `logits` the network's raw output.
- `dist` the action distribution.
- `state` the hidden state.

See also:

Please refer to *forward()* for more detailed explanation.

learn(*batch: Batch*, *batch_size: int*, *repeat: int*, ***kwargs: Any*) → Dict[str, List[float]]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

class tianshou.policy.**NPGPolicy**(*actor: Module*, *critic: Module*, *optim: Optimizer*, *dist_fn: Type[Distribution]*, *advantage_normalization: bool = True*, *optim_critic_iters: int = 5*, *actor_step_size: float = 0.5*, ***kwargs: Any*)

Bases: *A2CPolicy*

Implementation of Natural Policy Gradient.

https://proceedings.neurips.cc/paper/2001/file/4b86abe48d358ecf194c56c69108433e-Paper.pdf

Parameters

- **actor** (*torch.nn.Module*) – the actor network following the rules in *BasePolicy*. (s -> logits)
- **critic** (*torch.nn.Module*) – the critic network. (s -> V(s))
- **optim** (*torch.optim.Optimizer*) – the optimizer for actor and critic network.
- **dist_fn** (*Type[torch.distributions.Distribution]*) – distribution class for computing the action.
- **advantage_normalization** (*bool*) – whether to do per mini-batch advantage normalization. Default to True.
- **optim_critic_iters** (*int*) – Number of times to optimize critic network per update. Default to 5.
- **gae_lambda** (*float*) – in [0, 1], param for Generalized Advantage Estimation. Default to 0.95.
- **reward_normalization** (*bool*) – normalize estimated values to have std close to 1. Default to False.
- **max_batchsize** (*int*) – the maximum size of the batch when computing GAE, depends on the size of available memory and the memory cost of the model; should be as large as possible within the memory constraint. Default to 256.
- **action_scaling** (*bool*) – whether to map actions from range [-1, 1] to range [action_spaces.low, action_spaces.high]. Default to True.
- **action_bound_method** (*str*) – method to bound action to range [-1, 1], can be either "clip" (for simply clipping the action), "tanh" (for applying tanh squashing) for now, or empty string for no bounding. Default to "clip".
- **action_space** (*Optional[gym.Space]*) – env's action space, mandatory if you want to use option "action_scaling" or "action_bound_method". Default to None.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).
- **deterministic_eval** (*bool*) – whether to use deterministic action instead of stochastic action sampled by the policy. Default to False.

process_fn(*batch:* Batch, *buffer:* ReplayBuffer, *indices: ndarray*) → *Batch*

Compute the discounted returns for each transition.

$$G_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$$

where T is the terminal time step, γ is the discount factor, $\gamma \in [0, 1]$.

learn(*batch:* Batch, *batch_size: int*, *repeat: int*, ***kwargs: Any*) → Dict[str, List[float]]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

class tianshou.policy.**A2CPolicy**(*actor: Module*, *critic: Module*, *optim: Optimizer*, *dist_fn: Type[Distribution]*, *vf_coef: float = 0.5*, *ent_coef: float = 0.01*, *max_grad_norm: Optional[float] = None*, *gae_lambda: float = 0.95*, *max_batchsize: int = 256*, ***kwargs: Any*)

Bases: *PGPolicy*

Implementation of Synchronous Advantage Actor-Critic. arXiv:1602.01783.

Parameters

- **actor** (*torch.nn.Module*) – the actor network following the rules in *BasePolicy*. (s -> logits)
- **critic** (*torch.nn.Module*) – the critic network. (s -> V(s))
- **optim** (*torch.optim.Optimizer*) – the optimizer for actor and critic network.
- **dist_fn** (*Type[torch.distributions.Distribution]*) – distribution class for computing the action.
- **discount_factor** (*float*) – in [0, 1]. Default to 0.99.
- **vf_coef** (*float*) – weight for value loss. Default to 0.5.
- **ent_coef** (*float*) – weight for entropy loss. Default to 0.01.
- **max_grad_norm** (*float*) – clipping gradients in back propagation. Default to None.
- **gae_lambda** (*float*) – in [0, 1], param for Generalized Advantage Estimation. Default to 0.95.
- **reward_normalization** (*bool*) – normalize estimated values to have std close to 1. Default to False.
- **max_batchsize** (*int*) – the maximum size of the batch when computing GAE, depends on the size of available memory and the memory cost of the model; should be as large as possible within the memory constraint. Default to 256.
- **action_scaling** (*bool*) – whether to map actions from range [-1, 1] to range [action_spaces.low, action_spaces.high]. Default to True.
- **action_bound_method** (*str*) – method to bound action to range [-1, 1], can be either "clip" (for simply clipping the action), "tanh" (for applying tanh squashing) for now, or empty string for no bounding. Default to "clip".
- **action_space** (*Optional[gym.Space]*) – env's action space, mandatory if you want to use option "action_scaling" or "action_bound_method". Default to None.

- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).
- **deterministic_eval** (*bool*) – whether to use deterministic action instead of stochastic action sampled by the policy. Default to False.

See also:

Please refer to *BasePolicy* for more detailed explanation.

process_fn(*batch: Batch*, *buffer: ReplayBuffer*, *indices: ndarray*) → *Batch*

Compute the discounted returns for each transition.

$$G_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$$

where T is the terminal time step, γ is the discount factor, $\gamma \in [0, 1]$.

learn(*batch: Batch*, *batch_size: int*, *repeat: int*, ***kwargs: Any*) → Dict[str, List[float]]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

class tianshou.policy.**TRPOPolicy**(*actor: Module*, *critic: Module*, *optim: Optimizer*, *dist_fn: Type[Distribution]*, *max_kl: float = 0.01*, *backtrack_coeff: float = 0.8*, *max_backtracks: int = 10*, ***kwargs: Any*)

Bases: *NPGPolicy*

Implementation of Trust Region Policy Optimization. arXiv:1502.05477.

Parameters

- **actor** (*torch.nn.Module*) – the actor network following the rules in *BasePolicy*. (s -> logits)
- **critic** (*torch.nn.Module*) – the critic network. (s -> V(s))
- **optim** (*torch.optim.Optimizer*) – the optimizer for actor and critic network.
- **dist_fn** (*Type[torch.distributions.Distribution]*) – distribution class for computing the action.
- **advantage_normalization** (*bool*) – whether to do per mini-batch advantage normalization. Default to True.

- **optim_critic_iters** (*int*) – Number of times to optimize critic network per update. Default to 5.
- **max_kl** (*int*) – max kl-divergence used to constrain each actor network update. Default to 0.01.
- **backtrack_coeff** (*float*) – Coefficient to be multiplied by step size when constraints are not met. Default to 0.8.
- **max_backtracks** (*int*) – Max number of backtracking times in linesearch. Default to 10.
- **gae_lambda** (*float*) – in [0, 1], param for Generalized Advantage Estimation. Default to 0.95.
- **reward_normalization** (*bool*) – normalize estimated values to have std close to 1. Default to False.
- **max_batchsize** (*int*) – the maximum size of the batch when computing GAE, depends on the size of available memory and the memory cost of the model; should be as large as possible within the memory constraint. Default to 256.
- **action_scaling** (*bool*) – whether to map actions from range [-1, 1] to range [action_spaces.low, action_spaces.high]. Default to True.
- **action_bound_method** (*str*) – method to bound action to range [-1, 1], can be either "clip" (for simply clipping the action), "tanh" (for applying tanh squashing) for now, or empty string for no bounding. Default to "clip".
- **action_space** (*Optional[gym.Space]*) – env's action space, mandatory if you want to use option "action_scaling" or "action_bound_method". Default to None.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).
- **deterministic_eval** (*bool*) – whether to use deterministic action instead of stochastic action sampled by the policy. Default to False.

learn(*batch: Batch*, *batch_size: int*, *repeat: int*, ***kwargs: Any*) → Dict[str, List[float]]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

class tianshou.policy.**PPOPolicy**(*actor: Module*, *critic: Module*, *optim: Optimizer*, *dist_fn: Type[Distribution]*, *eps_clip: float = 0.2*, *dual_clip: Optional[float] = None*, *value_clip: bool = False*, *advantage_normalization: bool = True*, *recompute_advantage: bool = False*, ***kwargs: Any*)

Bases: *A2CPolicy*

Implementation of Proximal Policy Optimization. arXiv:1707.06347.

Parameters

- **actor** (*torch.nn.Module*) – the actor network following the rules in *BasePolicy*. (s -> logits)
- **critic** (*torch.nn.Module*) – the critic network. (s -> V(s))
- **optim** (*torch.optim.Optimizer*) – the optimizer for actor and critic network.
- **dist_fn** (*Type[torch.distributions.Distribution]*) – distribution class for computing the action.
- **discount_factor** (*float*) – in [0, 1]. Default to 0.99.
- **eps_clip** (*float*) – ϵ in L_{CLIP} in the original paper. Default to 0.2.
- **dual_clip** (*float*) – a parameter c mentioned in arXiv:1912.09729 Equ. 5, where c > 1 is a constant indicating the lower bound. Default to 5.0 (set None if you do not want to use it).
- **value_clip** (*bool*) – a parameter mentioned in arXiv:1811.02553v3 Sec. 4.1. Default to True.
- **advantage_normalization** (*bool*) – whether to do per mini-batch advantage normalization. Default to True.
- **recompute_advantage** (*bool*) – whether to recompute advantage every update repeat according to https://arxiv.org/pdf/2006.05990.pdf Sec. 3.5. Default to False.
- **vf_coef** (*float*) – weight for value loss. Default to 0.5.
- **ent_coef** (*float*) – weight for entropy loss. Default to 0.01.
- **max_grad_norm** (*float*) – clipping gradients in back propagation. Default to None.
- **gae_lambda** (*float*) – in [0, 1], param for Generalized Advantage Estimation. Default to 0.95.
- **reward_normalization** (*bool*) – normalize estimated values to have std close to 1, also normalize the advantage to Normal(0, 1). Default to False.
- **max_batchsize** (*int*) – the maximum size of the batch when computing GAE, depends on the size of available memory and the memory cost of the model; should be as large as possible within the memory constraint. Default to 256.
- **action_scaling** (*bool*) – whether to map actions from range [-1, 1] to range [action_spaces.low, action_spaces.high]. Default to True.
- **action_bound_method** (*str*) – method to bound action to range [-1, 1], can be either "clip" (for simply clipping the action), "tanh" (for applying tanh squashing) for now, or empty string for no bounding. Default to "clip".
- **action_space** (*Optional[gym.Space]*) – env's action space, mandatory if you want to use option "action_scaling" or "action_bound_method". Default to None.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).
- **deterministic_eval** (*bool*) – whether to use deterministic action instead of stochastic action sampled by the policy. Default to False.

See also:

Please refer to *BasePolicy* for more detailed explanation.

process_fn(*batch:* Batch, *buffer:* ReplayBuffer, *indices: ndarray*) → *Batch*

Compute the discounted returns for each transition.

$$G_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$$

where T is the terminal time step, γ is the discount factor, $\gamma \in [0, 1]$.

learn(*batch:* Batch, *batch_size: int*, *repeat: int*, ***kwargs: Any*) → Dict[str, List[float]]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

Off-policy

class tianshou.policy.**DDPGPolicy**(*actor: ~typing.Optional[~torch.nn.modules.module.Module], actor_optim: ~typing.Optional[~torch.optim.optimizer.Optimizer], critic: ~typing.Optional[~torch.nn.modules.module.Module], critic_optim: ~typing.Optional[~torch.optim.optimizer.Optimizer], tau: float = 0.005, gamma: float = 0.99, exploration_noise: ~typing.Optional[~tianshou.exploration.random.BaseNoise] = <tianshou.exploration.random.GaussianNoise object>, reward_normalization: bool = False, estimation_step: int = 1, action_scaling: bool = True, action_bound_method: str = 'clip', **kwargs: ~typing.Any*)

Bases: *BasePolicy*

Implementation of Deep Deterministic Policy Gradient. arXiv:1509.02971.

Parameters

- **actor** (*torch.nn.Module*) – the actor network following the rules in *BasePolicy*. (s -> logits)
- **actor_optim** (*torch.optim.Optimizer*) – the optimizer for actor network.
- **critic** (*torch.nn.Module*) – the critic network. (s, a -> Q(s, a))

- **critic_optim** (*torch.optim.Optimizer*) – the optimizer for critic network.
- **tau** (*float*) – param for soft update of the target network. Default to 0.005.
- **gamma** (*float*) – discount factor, in [0, 1]. Default to 0.99.
- **exploration_noise** (BaseNoise) – the exploration noise, add to the action. Default to `GaussianNoise(sigma=0.1)`.
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1), Default to False.
- **estimation_step** (*int*) – the number of steps to look ahead. Default to 1.
- **action_scaling** (*bool*) – whether to map actions from range [-1, 1] to range [action_spaces.low, action_spaces.high]. Default to True.
- **action_bound_method** (*str*) – method to bound action to range [-1, 1], can be either "clip" (for simply clipping the action) or empty string for no bounding. Default to "clip".
- **action_space** (*Optional[gym.Space]*) – env's action space, mandatory if you want to use option "action_scaling" or "action_bound_method". Default to None.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *BasePolicy* for more detailed explanation.

set_exp_noise(*noise: Optional[BaseNoise]*) → None

Set the exploration noise.

train(*mode: bool = True*) → *DDPGPolicy*

Set the module in training mode, except for the target network.

sync_weight() → None

Soft-update the weight for the target network.

process_fn(*batch:* Batch, *buffer:* ReplayBuffer, *indices: ndarray*) → *Batch*

Pre-process the data from the provided replay buffer.

Used in `update()`. Check out *policy.process_fn* for more information.

forward(*batch:* Batch, *state: Optional[Union[dict,* Batch*, ndarray]] = None*, *model: str = 'actor'*, *input: str = 'obs'*, ***kwargs: Any*) → *Batch*

Compute action over the given batch data.

Returns

A *Batch* which has 2 keys:

- `act` the action.
- `state` the hidden state.

See also:

Please refer to *forward()* for more detailed explanation.

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

exploration_noise(*act: Union[ndarray,* Batch*]*, *batch:* Batch) → Union[ndarray, *Batch*]

Modify the action from policy.forward with exploration noise.

Parameters

- **act** – a data batch or numpy.ndarray which is the action taken by policy.forward.
- **batch** – the input batch for policy.forward, kept for advanced usage.

Returns

action in the same form of input "act" but with added exploration noise.

training: bool

class tianshou.policy.**TD3Policy**(*actor: ~torch.nn.modules.module.Module*, *actor_optim: ~torch.optim.optimizer.Optimizer*, *critic1: ~torch.nn.modules.module.Module*, *critic1_optim: ~torch.optim.optimizer.Optimizer*, *critic2: ~torch.nn.modules.module.Module*, *critic2_optim: ~torch.optim.optimizer.Optimizer*, *tau: float = 0.005*, *gamma: float = 0.99*, *exploration_noise: ~typing.Optional[~tianshou.exploration.random.BaseNoise] = <tianshou.exploration.random.GaussianNoise object>*, *policy_noise: float = 0.2*, *update_actor_freq: int = 2*, *noise_clip: float = 0.5*, *reward_normalization: bool = False*, *estimation_step: int = 1*, ***kwargs: ~typing.Any*)

Bases: *DDPGPolicy*

Implementation of TD3, arXiv:1802.09477.

Parameters

- **actor** (*torch.nn.Module*) – the actor network following the rules in *BasePolicy*. (s -> logits)
- **actor_optim** (*torch.optim.Optimizer*) – the optimizer for actor network.
- **critic1** (*torch.nn.Module*) – the first critic network. (s, a -> Q(s, a))
- **critic1_optim** (*torch.optim.Optimizer*) – the optimizer for the first critic network.
- **critic2** (*torch.nn.Module*) – the second critic network. (s, a -> Q(s, a))
- **critic2_optim** (*torch.optim.Optimizer*) – the optimizer for the second critic network.
- **tau** (*float*) – param for soft update of the target network. Default to 0.005.
- **gamma** (*float*) – discount factor, in [0, 1]. Default to 0.99.

- **exploration_noise** (*float*) – the exploration noise, add to the action. Default to `GaussianNoise(sigma=0.1)`
- **policy_noise** (*float*) – the noise used in updating policy network. Default to 0.2.
- **update_actor_freq** (*int*) – the update frequency of actor network. Default to 2.
- **noise_clip** (*float*) – the clipping range used in updating policy network. Default to 0.5.
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1). Default to False.
- **action_scaling** (*bool*) – whether to map actions from range [-1, 1] to range [action_spaces.low, action_spaces.high]. Default to True.
- **action_bound_method** (*str*) – method to bound action to range [-1, 1], can be either "clip" (for simply clipping the action) or empty string for no bounding. Default to "clip".
- **action_space** (*Optional[gym.Space]*) – env's action space, mandatory if you want to use option "action_scaling" or "action_bound_method". Default to None.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *BasePolicy* for more detailed explanation.

train(*mode: bool = True*) → *TD3Policy*

Set the module in training mode, except for the target network.

sync_weight() → None

Soft-update the weight for the target network.

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

actor: torch.nn.Module

actor_optim: torch.optim.Optimizer

critic: torch.nn.Module

critic_optim: torch.optim.Optimizer

class tianshou.policy.**SACPolicy**(*actor: Module*, *actor_optim: Optimizer*, *critic1: Module*, *critic1_optim: Optimizer*, *critic2: Module*, *critic2_optim: Optimizer*, *tau: float = 0.005*, *gamma: float = 0.99*, *alpha: Union[float, Tuple[float, Tensor, Optimizer]] = 0.2*, *reward_normalization: bool = False*, *estimation_step: int = 1*, *exploration_noise: Optional[BaseNoise] = None*, *deterministic_eval: bool = True*, ***kwargs: Any*)

Bases: *DDPGPolicy*

Implementation of Soft Actor-Critic. arXiv:1812.05905.

Parameters

- **actor** (*torch.nn.Module*) – the actor network following the rules in *BasePolicy*. (s -> logits)
- **actor_optim** (*torch.optim.Optimizer*) – the optimizer for actor network.
- **critic1** (*torch.nn.Module*) – the first critic network. (s, a -> Q(s, a))
- **critic1_optim** (*torch.optim.Optimizer*) – the optimizer for the first critic network.
- **critic2** (*torch.nn.Module*) – the second critic network. (s, a -> Q(s, a))
- **critic2_optim** (*torch.optim.Optimizer*) – the optimizer for the second critic network.
- **tau** (*float*) – param for soft update of the target network. Default to 0.005.
- **gamma** (*float*) – discount factor, in [0, 1]. Default to 0.99.
- **alpha** (*(float, torch.Tensor, torch.optim.Optimizer) or float*) – entropy regularization coefficient. Default to 0.2. If a tuple (target_entropy, log_alpha, alpha_optim) is provided, then alpha is automatically tuned.
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1). Default to False.
- **exploration_noise** (BaseNoise) – add a noise to action for exploration. Default to None. This is useful when solving hard-exploration problem.
- **deterministic_eval** (*bool*) – whether to use deterministic action (mean of Gaussian policy) instead of stochastic action sampled by the policy. Default to True.
- **action_scaling** (*bool*) – whether to map actions from range [-1, 1] to range [action_spaces.low, action_spaces.high]. Default to True.
- **action_bound_method** (*str*) – method to bound action to range [-1, 1], can be either "clip" (for simply clipping the action) or empty string for no bounding. Default to "clip".
- **action_space** (*Optional[gym.Space]*) – env's action space, mandatory if you want to use option "action_scaling" or "action_bound_method". Default to None.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *BasePolicy* for more detailed explanation.

actor: torch.nn.Module

actor_optim: torch.optim.Optimizer

train(*mode: bool = True*) → *SACPolicy*

Set the module in training mode, except for the target network.

sync_weight() → None

Soft-update the weight for the target network.

forward(*batch:* Batch, *state: Optional[Union[dict,* Batch*, ndarray]] = None*, *input: str = 'obs'*, ***kwargs: Any*) → *Batch*

Compute action over the given batch data.

Returns

A *Batch* which has 2 keys:

- `act` the action.
- `state` the hidden state.

See also:

Please refer to *forward()* for more detailed explanation.

training: bool

critic: torch.nn.Module

critic_optim: torch.optim.Optimizer

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

class tianshou.policy.**REDQPolicy**(*actor: Module*, *actor_optim: Optimizer*, *critics: Module*, *critics_optim: Optimizer*, *ensemble_size: int = 10*, *subset_size: int = 2*, *tau: float = 0.005*, *gamma: float = 0.99*, *alpha: Union[float, Tuple[float, Tensor, Optimizer]] = 0.2*, *reward_normalization: bool = False*, *estimation_step: int = 1*, *actor_delay: int = 20*, *exploration_noise: Optional[*BaseNoise*] = None*, *deterministic_eval: bool = True*, *target_mode: str = 'min'*, ***kwargs: Any*)

Bases: *DDPGPolicy*

Implementation of REDQ. arXiv:2101.05982.

Parameters

- **actor** (*torch.nn.Module*) – the actor network following the rules in *BasePolicy*. (s -> logits)
- **actor_optim** (*torch.optim.Optimizer*) – the optimizer for actor network.

- **critics** (*torch.nn.Module*) – critic ensemble networks.
- **critics_optim** (*torch.optim.Optimizer*) – the optimizer for the critic networks.
- **ensemble_size** (*int*) – Number of sub-networks in the critic ensemble. Default to 10.
- **subset_size** (*int*) – Number of networks in the subset. Default to 2.
- **tau** (*float*) – param for soft update of the target network. Default to 0.005.
- **gamma** (*float*) – discount factor, in [0, 1]. Default to 0.99.
- **alpha** (*(float, torch.Tensor, torch.optim.Optimizer) or float*) – entropy regularization coefficient. Default to 0.2. If a tuple (target_entropy, log_alpha, alpha_optim) is provided, then alpha is automatically tuned.
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1). Default to False.
- **actor_delay** (*int*) – Number of critic updates before an actor update. Default to 20.
- **exploration_noise** (BaseNoise) – add a noise to action for exploration. Default to None. This is useful when solving hard-exploration problem.
- **deterministic_eval** (*bool*) – whether to use deterministic action (mean of Gaussian policy) instead of stochastic action sampled by the policy. Default to True.
- **target_mode** (*str*) – methods to integrate critic values in the subset, currently support minimum and average. Default to min.
- **action_scaling** (*bool*) – whether to map actions from range [-1, 1] to range [action_spaces.low, action_spaces.high]. Default to True.
- **action_bound_method** (*str*) – method to bound action to range [-1, 1], can be either "clip" (for simply clipping the action) or empty string for no bounding. Default to "clip".
- **action_space** (*Optional[gym.Space]*) – env's action space, mandatory if you want to use option "action_scaling" or "action_bound_method". Default to None.

See also:

Please refer to *BasePolicy* for more detailed explanation.

actor: torch.nn.Module

actor_optim: torch.optim.Optimizer

train(*mode: bool = True*) → *REDQPolicy*

Set the module in training mode, except for the target network.

sync_weight() → None

Soft-update the weight for the target network.

forward(*batch:* Batch, *state: Optional[Union[dict,* Batch*, ndarray]] = None*, *input: str = 'obs'*, ***kwargs: Any*) → *Batch*

Compute action over the given batch data.

Returns

A *Batch* which has 2 keys:

- `act` the action.
- `state` the hidden state.

See also:

Please refer to *forward()* for more detailed explanation.

training: bool

critic: torch.nn.Module

critic_optim: torch.optim.Optimizer

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

class tianshou.policy.**DiscreteSACPolicy**(*actor: Module*, *actor_optim: Optimizer*, *critic1: Module*, *critic1_optim: Optimizer*, *critic2: Module*, *critic2_optim: Optimizer*, *tau: float = 0.005*, *gamma: float = 0.99*, *alpha: Union[float, Tuple[float, Tensor, Optimizer]] = 0.2*, *reward_normalization: bool = False*, *estimation_step: int = 1*, ***kwargs: Any*)

Bases: *SACPolicy*

Implementation of SAC for Discrete Action Settings. arXiv:1910.07207.

Parameters

- **actor** (*torch.nn.Module*) – the actor network following the rules in *BasePolicy*. (s -> logits)
- **actor_optim** (*torch.optim.Optimizer*) – the optimizer for actor network.
- **critic1** (*torch.nn.Module*) – the first critic network. (s -> Q(s))
- **critic1_optim** (*torch.optim.Optimizer*) – the optimizer for the first critic network.
- **critic2** (*torch.nn.Module*) – the second critic network. (s -> Q(s))
- **critic2_optim** (*torch.optim.Optimizer*) – the optimizer for the second critic network.
- **tau** (*float*) – param for soft update of the target network. Default to 0.005.
- **gamma** (*float*) – discount factor, in [0, 1]. Default to 0.99.
- **alpha** (*(float, torch.Tensor, torch.optim.Optimizer) or float*) – entropy regularization coefficient. Default to 0.2. If a tuple (target_entropy, log_alpha, alpha_optim) is provided, the alpha is automatically tuned.
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1). Default to False.

- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *BasePolicy* for more detailed explanation.

forward(*batch:* Batch, *state: Optional[Union[dict,* Batch*, ndarray]] = None*, *input: str = 'obs'*, ***kwargs: Any*) → *Batch*

Compute action over the given batch data.

Returns

A *Batch* which has 2 keys:

- `act` the action.
- `state` the hidden state.

See also:

Please refer to *forward()* for more detailed explanation.

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

exploration_noise(*act: Union[ndarray,* Batch*]*, *batch:* Batch) → Union[ndarray, *Batch*]

Modify the action from policy.forward with exploration noise.

Parameters

- **act** – a data batch or numpy.ndarray which is the action taken by policy.forward.
- **batch** – the input batch for policy.forward, kept for advanced usage.

Returns

action in the same form of input "act" but with added exploration noise.

training: bool

actor: torch.nn.Module

actor_optim: torch.optim.Optimizer

critic: torch.nn.Module

critic_optim: torch.optim.Optimizer

1.11.3 Imitation

class tianshou.policy.**ImitationPolicy**(*model: Module*, *optim: Optimizer*, ***kwargs: Any*)

Bases: *BasePolicy*

Implementation of vanilla imitation learning.

Parameters

- **model** (*torch.nn.Module*) – a model following the rules in *BasePolicy*. (s -> a)
- **optim** (*torch.optim.Optimizer*) – for optimizing the model.
- **action_space** (*gym.Space*) – env's action space.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *BasePolicy* for more detailed explanation.

forward(*batch:* Batch, *state: Optional[Union[dict,* Batch*, ndarray]] = None*, ***kwargs: Any*) → *Batch*

Compute action over the given batch data.

Returns

A *Batch* which MUST have the following keys:

- `act` an numpy.ndarray or a torch.Tensor, the action over given batch data.
- `state` a dict, an numpy.ndarray or a torch.Tensor, the internal state of the policy, `None` as default.

Other keys are user-defined. It depends on the algorithm. For example,

```
# some code
return Batch(logits=..., act=..., state=None, dist=...)
```

The keyword `policy` is reserved and the corresponding data will be stored into the replay buffer. For instance,

```
# some code
return Batch(..., policy=Batch(log_prob=dist.log_prob(act)))
# and in the sampled data batch, you can directly use
# batch.policy.log_prob to get your data.
```

Note: In continuous action space, you should do another step "map_action" to get the real action:

```
act = policy(batch).act  # doesn't map to the target action range
act = policy.map_action(act, batch)
```

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

class tianshou.policy.**BCQPolicy**(*actor: Module*, *actor_optim: Optimizer*, *critic1: Module*, *critic1_optim: Optimizer*, *critic2: Module*, *critic2_optim: Optimizer*, *vae:* VAE, *vae_optim: Optimizer*, *device: Union[str, device] = 'cpu'*, *gamma: float = 0.99*, *tau: float = 0.005*, *lmbda: float = 0.75*, *forward_sampled_times: int = 100*, *num_sampled_action: int = 10*, ***kwargs: Any*)

Bases: *BasePolicy*

Implementation of BCQ algorithm. arXiv:1812.02900.

Parameters

- **actor** (Perturbation) – the actor perturbation. (s, a -> perturbed a)
- **actor_optim** (*torch.optim.Optimizer*) – the optimizer for actor network.
- **critic1** (*torch.nn.Module*) – the first critic network. (s, a -> Q(s, a))
- **critic1_optim** (*torch.optim.Optimizer*) – the optimizer for the first critic network.
- **critic2** (*torch.nn.Module*) – the second critic network. (s, a -> Q(s, a))
- **critic2_optim** (*torch.optim.Optimizer*) – the optimizer for the second critic network.
- **vae** (VAE) – the VAE network, generating actions similar to those in batch. (s, a -> generated a)
- **vae_optim** (*torch.optim.Optimizer*) – the optimizer for the VAE network.
- **device** (*Union[str, torch.device]*) – which device to create this model on. Default to "cpu".
- **gamma** (*float*) – discount factor, in [0, 1]. Default to 0.99.
- **tau** (*float*) – param for soft update of the target network. Default to 0.005.
- **lmbda** (*float*) – param for Clipped Double Q-learning. Default to 0.75.
- **forward_sampled_times** (*int*) – the number of sampled actions in forward function. The policy samples many actions and takes the action with the max value. Default to 100.
- **num_sampled_action** (*int*) – the number of sampled actions in calculating target Q. The algorithm samples several actions using VAE, and perturbs each action to get the target Q. Default to 10.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *BasePolicy* for more detailed explanation.

train(*mode: bool = True*) → *BCQPolicy*

Set the module in training mode, except for the target network.

forward(*batch:* Batch, *state: Optional[Union[dict,* Batch*, ndarray]] = None*, ***kwargs: Any*) → *Batch*

Compute action over the given batch data.

sync_weight() → None

Soft-update the weight for the target network.

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

class tianshou.policy.**CQLPolicy**(*actor:* ActorProb, *actor_optim: Optimizer*, *critic1: Module*, *critic1_optim: Optimizer*, *critic2: Module*, *critic2_optim: Optimizer*, *cql_alpha_lr: float = 0.0001*, *cql_weight: float = 1.0*, *tau: float = 0.005*, *gamma: float = 0.99*, *alpha: Union[float, Tuple[float, Tensor, Optimizer]] = 0.2*, *temperature: float = 1.0*, *with_lagrange: bool = True*, *lagrange_threshold: float = 10.0*, *min_action: float = -1.0*, *max_action: float = 1.0*, *num_repeat_actions: int = 10*, *alpha_min: float = 0.0*, *alpha_max: float = 1000000.0*, *clip_grad: float = 1.0*, *device: Union[str, device] = 'cpu'*, ***kwargs: Any*)

Bases: *SACPolicy*

Implementation of CQL algorithm. arXiv:2006.04779.

Parameters

- **actor** (ActorProb) – the actor network following the rules in *BasePolicy*. (s -> a)
- **actor_optim** (*torch.optim.Optimizer*) – the optimizer for actor network.
- **critic1** (*torch.nn.Module*) – the first critic network. (s, a -> Q(s, a))
- **critic1_optim** (*torch.optim.Optimizer*) – the optimizer for the first critic network.
- **critic2** (*torch.nn.Module*) – the second critic network. (s, a -> Q(s, a))
- **critic2_optim** (*torch.optim.Optimizer*) – the optimizer for the second critic network.
- **cql_alpha_lr** (*float*) – the learning rate of cql_log_alpha. Default to 1e-4.

- **cql_weight** (*float*) – the value of alpha. Default to 1.0.
- **tau** (*float*) – param for soft update of the target network. Default to 0.005.
- **gamma** (*float*) – discount factor, in [0, 1]. Default to 0.99.
- **alpha** (*(float, torch.Tensor, torch.optim.Optimizer) or float*) – entropy regularization coefficient. Default to 0.2. If a tuple (target_entropy, log_alpha, alpha_optim) is provided, then alpha is automatically tuned.
- **temperature** (*float*) – the value of temperature. Default to 1.0.
- **with_lagrange** (*bool*) – whether to use Lagrange. Default to True.
- **lagrange_threshold** (*float*) – the value of tau in CQL(Lagrange). Default to 10.0.
- **min_action** (*float*) – The minimum value of each dimension of action. Default to -1.0.
- **max_action** (*float*) – The maximum value of each dimension of action. Default to 1.0.
- **num_repeat_actions** (*int*) – The number of times the action is repeated when calculating log-sum-exp. Default to 10.
- **alpha_min** (*float*) – lower bound for clipping cql_alpha. Default to 0.0.
- **alpha_max** (*float*) – upper bound for clipping cql_alpha. Default to 1e6.
- **clip_grad** (*float*) – clip_grad for updating critic network. Default to 1.0.
- **device** (*Union[str, torch.device]*) – which device to create this model on. Default to "cpu".
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *BasePolicy* for more detailed explanation.

train(*mode: bool = True*) → *CQLPolicy*

Set the module in training mode, except for the target network.

sync_weight() → None

Soft-update the weight for the target network.

actor_pred(*obs: Tensor*) → Tuple[Tensor, Tensor]

calc_actor_loss(*obs: Tensor*) → Tuple[Tensor, Tensor]

calc_pi_values(*obs_pi: Tensor*, *obs_to_pred: Tensor*) → Tuple[Tensor, Tensor]

calc_random_values(*obs: Tensor*, *act: Tensor*) → Tuple[Tensor, Tensor]

process_fn(*batch:* Batch, *buffer:* ReplayBuffer, *indices: ndarray*) → *Batch*

Pre-process the data from the provided replay buffer.

Used in `update()`. Check out *policy.process_fn* for more information.

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

actor: torch.nn.Module

actor_optim: torch.optim.Optimizer

critic: torch.nn.Module

critic_optim: torch.optim.Optimizer

class tianshou.policy.**TD3BCPolicy**(*actor: ~torch.nn.modules.module.Module*, *actor_optim: ~torch.optim.optimizer.Optimizer*, *critic1: ~torch.nn.modules.module.Module*, *critic1_optim: ~torch.optim.optimizer.Optimizer*, *critic2: ~torch.nn.modules.module.Module*, *critic2_optim: ~torch.optim.optimizer.Optimizer*, *tau: float = 0.005*, *gamma: float = 0.99*, *exploration_noise: ~typing.Optional[~tianshou.exploration.random.BaseNoise] = <tianshou.exploration.random.GaussianNoise object>*, *policy_noise: float = 0.2*, *update_actor_freq: int = 2*, *noise_clip: float = 0.5*, *alpha: float = 2.5*, *reward_normalization: bool = False*, *estimation_step: int = 1*, ***kwargs: ~typing.Any*)

Bases: *TD3Policy*

Implementation of TD3+BC. arXiv:2106.06860.

Parameters

- **actor** (*torch.nn.Module*) – the actor network following the rules in *BasePolicy*. (s -> logits)
- **actor_optim** (*torch.optim.Optimizer*) – the optimizer for actor network.
- **critic1** (*torch.nn.Module*) – the first critic network. (s, a -> Q(s, a))
- **critic1_optim** (*torch.optim.Optimizer*) – the optimizer for the first critic network.
- **critic2** (*torch.nn.Module*) – the second critic network. (s, a -> Q(s, a))
- **critic2_optim** (*torch.optim.Optimizer*) – the optimizer for the second critic network.
- **tau** (*float*) – param for soft update of the target network. Default to 0.005.
- **gamma** (*float*) – discount factor, in [0, 1]. Default to 0.99.
- **exploration_noise** (*float*) – the exploration noise, add to the action. Default to `GaussianNoise(sigma=0.1)`
- **policy_noise** (*float*) – the noise used in updating policy network. Default to 0.2.

- **update_actor_freq** (*int*) – the update frequency of actor network. Default to 2.
- **noise_clip** (*float*) – the clipping range used in updating policy network. Default to 0.5.
- **alpha** (*float*) – the value of alpha, which controls the weight for TD3 learning relative to behavior cloning.
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1). Default to False.
- **action_scaling** (*bool*) – whether to map actions from range [-1, 1] to range [action_spaces.low, action_spaces.high]. Default to True.
- **action_bound_method** (*str*) – method to bound action to range [-1, 1], can be either "clip" (for simply clipping the action) or empty string for no bounding. Default to "clip".
- **action_space** (*Optional[gym.Space]*) – env's action space, mandatory if you want to use option "action_scaling" or "action_bound_method". Default to None.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *BasePolicy* for more detailed explanation.

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

actor: torch.nn.Module

actor_optim: torch.optim.Optimizer

critic: torch.nn.Module

critic_optim: torch.optim.Optimizer

class tianshou.policy.**DiscreteBCQPolicy**(*model: Module*, *imitator: Module*, *optim: Optimizer*, *discount_factor: float = 0.99*, *estimation_step: int = 1*, *target_update_freq: int = 8000*, *eval_eps: float = 0.001*, *unlikely_action_threshold: float = 0.3*, *imitation_logits_penalty: float = 0.01*, *reward_normalization: bool = False*, ***kwargs: Any*)

Bases: *DQNPolicy*

Implementation of discrete BCQ algorithm. arXiv:1910.01708.

Parameters

- **model** (*torch.nn.Module*) – a model following the rules in *BasePolicy*. (s -> q_value)
- **imitator** (*torch.nn.Module*) – a model following the rules in *BasePolicy*. (s -> imitation_logits)
- **optim** (*torch.optim.Optimizer*) – a torch.optim for optimizing the model.
- **discount_factor** (*float*) – in [0, 1].
- **estimation_step** (*int*) – the number of steps to look ahead. Default to 1.
- **target_update_freq** (*int*) – the target network update frequency.
- **eval_eps** (*float*) – the epsilon-greedy noise added in evaluation.
- **unlikely_action_threshold** (*float*) – the threshold (tau) for unlikely actions, as shown in Equ. (17) in the paper. Default to 0.3.
- **imitation_logits_penalty** (*float*) – regularization weight for imitation logits. Default to 1e-2.
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1). Default to False.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *BasePolicy* for more detailed explanation.

train(*mode: bool = True*) → *DiscreteBCQPolicy*

Set the module in training mode, except for the target network.

forward(*batch:* Batch, *state: Optional[Union[dict,* Batch*, ndarray]] = None*, *input: str = 'obs'*, ***kwargs: Any*) → *Batch*

Compute action over the given batch data.

If you need to mask the action, please add a "mask" into batch.obs, for example, if we have an environment that has "0/1/2" three actions:

```
batch == Batch(
    obs=Batch(
        obs="original obs, with batch_size=1 for demonstration",
        mask=np.array([[False, True, False]]),
        # action 1 is available
        # action 0 and 2 are unavailable
    ),
    ...
)
```

Returns

A *Batch* which has 3 keys:

- `act` the action.
- `logits` the network's raw output.

- `state` the hidden state.

See also:

Please refer to *forward()* for more detailed explanation.

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

class tianshou.policy.**DiscreteCQLPolicy**(*model: Module*, *optim: Optimizer*, *discount_factor: float = 0.99*, *num_quantiles: int = 200*, *estimation_step: int = 1*, *target_update_freq: int = 0*, *reward_normalization: bool = False*, *min_q_weight: float = 10.0*, ***kwargs: Any*)

Bases: *QRDQNPolicy*

Implementation of discrete Conservative Q-Learning algorithm. arXiv:2006.04779.

Parameters

- **model** (*torch.nn.Module*) – a model following the rules in *BasePolicy*. (s -> logits)
- **optim** (*torch.optim.Optimizer*) – a torch.optim for optimizing the model.
- **discount_factor** (*float*) – in [0, 1].
- **num_quantiles** (*int*) – the number of quantile midpoints in the inverse cumulative distribution function of the value. Default to 200.
- **estimation_step** (*int*) – the number of steps to look ahead. Default to 1.
- **target_update_freq** (*int*) – the target network update frequency (0 if you do not use the target network).
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1). Default to False.
- **min_q_weight** (*float*) – the weight for the cql loss.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *QRDQNPolicy* for more detailed explanation.

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

class tianshou.policy.**DiscreteCRRPolicy**(*actor: Module*, *critic: Module*, *optim: Optimizer*, *discount_factor: float = 0.99*, *policy_improvement_mode: str = 'exp'*, *ratio_upper_bound: float = 20.0*, *beta: float = 1.0*, *min_q_weight: float = 10.0*, *target_update_freq: int = 0*, *reward_normalization: bool = False*, ***kwargs: Any*)

Bases: *PGPolicy*

Implementation of discrete Critic Regularized Regression. arXiv:2006.15134.

Parameters

- **actor** (*torch.nn.Module*) – the actor network following the rules in *BasePolicy*. (s -> logits)
- **critic** (*torch.nn.Module*) – the action-value critic (i.e., Q function) network. (s -> Q(s, *))
- **optim** (*torch.optim.Optimizer*) – a torch.optim for optimizing the model.
- **discount_factor** (*float*) – in [0, 1]. Default to 0.99.
- **policy_improvement_mode** (*str*) – type of the weight function f. Possible values: "binary"/"exp"/"all". Default to "exp".
- **ratio_upper_bound** (*float*) – when policy_improvement_mode is "exp", the value of the exp function is upper-bounded by this parameter. Default to 20.
- **beta** (*float*) – when policy_improvement_mode is "exp", this is the denominator of the exp function. Default to 1.
- **min_q_weight** (*float*) – weight for CQL loss/regularizer. Default to 10.
- **target_update_freq** (*int*) – the target network update frequency (0 if you do not use the target network). Default to 0.
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1). Default to False.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *PGPolicy* for more detailed explanation.

sync_weight() → None

training: bool

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

class tianshou.policy.**GAILPolicy**(*actor: Module*, *critic: Module*, *optim: Optimizer*, *dist_fn: Type[Distribution]*, *expert_buffer:* ReplayBuffer, *disc_net: Module*, *disc_optim: Optimizer*, *disc_update_num: int = 4*, *eps_clip: float = 0.2*, *dual_clip: Optional[float] = None*, *value_clip: bool = False*, *advantage_normalization: bool = True*, *recompute_advantage: bool = False*, ***kwargs: Any*)

Bases: *PPOPolicy*

Implementation of Generative Adversarial Imitation Learning. arXiv:1606.03476.

Parameters

- **actor** (*torch.nn.Module*) – the actor network following the rules in *BasePolicy*. (s -> logits)
- **critic** (*torch.nn.Module*) – the critic network. (s -> V(s))
- **optim** (*torch.optim.Optimizer*) – the optimizer for actor and critic network.
- **dist_fn** (*Type[torch.distributions.Distribution]*) – distribution class for computing the action.
- **expert_buffer** (ReplayBuffer) – the replay buffer contains expert experience.
- **disc_net** (*torch.nn.Module*) – the discriminator network with input dim equals state dim plus action dim and output dim equals 1.
- **disc_optim** (*torch.optim.Optimizer*) – the optimizer for the discriminator network.
- **disc_update_num** (*int*) – the number of discriminator grad steps per model grad step. Default to 4.
- **discount_factor** (*float*) – in [0, 1]. Default to 0.99.
- **eps_clip** (*float*) – ϵ in L_{CLIP} in the original paper. Default to 0.2.

- **dual_clip** (*float*) – a parameter c mentioned in arXiv:1912.09729 Equ. 5, where c > 1 is a constant indicating the lower bound. Default to 5.0 (set None if you do not want to use it).
- **value_clip** (*bool*) – a parameter mentioned in arXiv:1811.02553 Sec. 4.1. Default to True.
- **advantage_normalization** (*bool*) – whether to do per mini-batch advantage normalization. Default to True.
- **recompute_advantage** (*bool*) – whether to recompute advantage every update repeat according to https://arxiv.org/pdf/2006.05990.pdf Sec. 3.5. Default to False.
- **vf_coef** (*float*) – weight for value loss. Default to 0.5.
- **ent_coef** (*float*) – weight for entropy loss. Default to 0.01.
- **max_grad_norm** (*float*) – clipping gradients in back propagation. Default to None.
- **gae_lambda** (*float*) – in [0, 1], param for Generalized Advantage Estimation. Default to 0.95.
- **reward_normalization** (*bool*) – normalize estimated values to have std close to 1, also normalize the advantage to Normal(0, 1). Default to False.
- **max_batchsize** (*int*) – the maximum size of the batch when computing GAE, depends on the size of available memory and the memory cost of the model; should be as large as possible within the memory constraint. Default to 256.
- **action_scaling** (*bool*) – whether to map actions from range [-1, 1] to range [action_spaces.low, action_spaces.high]. Default to True.
- **action_bound_method** (*str*) – method to bound action to range [-1, 1], can be either "clip" (for simply clipping the action), "tanh" (for applying tanh squashing) for now, or empty string for no bounding. Default to "clip".
- **action_space** (*Optional[gym.Space]*) – env's action space, mandatory if you want to use option "action_scaling" or "action_bound_method". Default to None.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).
- **deterministic_eval** (*bool*) – whether to use deterministic action instead of stochastic action sampled by the policy. Default to False.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *PPOPolicy* for more detailed explanation.

process_fn(*batch:* Batch, *buffer:* ReplayBuffer, *indices: ndarray*) → *Batch*

Pre-process the data from the provided replay buffer.

Used in `update()`. Check out *policy.process_fn* for more information.

disc(*batch:* Batch) → Tensor

learn(*batch:* Batch, *batch_size: int*, *repeat: int*, ***kwargs: Any*) → Dict[str, List[float]]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

1.11.4 Model-based

class tianshou.policy.**PSRLPolicy**(*trans_count_prior: ndarray*, *rew_mean_prior: ndarray*, *rew_std_prior: ndarray*, *discount_factor: float = 0.99*, *epsilon: float = 0.01*, *add_done_loop: bool = False*, ***kwargs: Any*)

Bases: *BasePolicy*

Implementation of Posterior Sampling Reinforcement Learning.

Reference: Strens M. A Bayesian framework for reinforcement learning [C] //ICML. 2000, 2000: 943-950.

Parameters

- **trans_count_prior** (*np.ndarray*) – dirichlet prior (alphas), with shape (n_state, n_action, n_state).
- **rew_mean_prior** (*np.ndarray*) – means of the normal priors of rewards, with shape (n_state, n_action).
- **rew_std_prior** (*np.ndarray*) – standard deviations of the normal priors of rewards, with shape (n_state, n_action).
- **discount_factor** (*float*) – in [0, 1].
- **epsilon** (*float*) – for precision control in value iteration.
- **add_done_loop** (*bool*) – whether to add an extra self-loop for the terminal state in MDP. Default to False.

See also:

Please refer to *BasePolicy* for more detailed explanation.

forward(*batch:* Batch, *state: Optional[Union[dict,* Batch*, ndarray]] = None*, ***kwargs: Any*) → *Batch*

Compute action over the given batch data with PSRL model.

Returns

A *Batch* with "act" key containing the action.

See also:

Please refer to *forward()* for more detailed explanation.

learn(*batch: Batch*, **args: Any*, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns

A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

class tianshou.policy.**ICMPolicy**(*policy: BasePolicy*, *model: IntrinsicCuriosityModule*, *optim: Optimizer*, *lr_scale: float*, *reward_scale: float*, *forward_loss_weight: float*, ***kwargs: Any*)

Bases: *BasePolicy*

Implementation of Intrinsic Curiosity Module. arXiv:1705.05363.

Parameters

- **policy** (*BasePolicy*) – a base policy to add ICM to.
- **model** (*IntrinsicCuriosityModule*) – the ICM model.
- **optim** (*torch.optim.Optimizer*) – a torch.optim for optimizing the model.
- **lr_scale** (*float*) – the scaling factor for ICM learning.
- **forward_loss_weight** (*float*) – the weight for forward model loss.
- **lr_scheduler** – a learning rate scheduler that adjusts the learning rate in optimizer in each policy.update(). Default to None (no lr_scheduler).

See also:

Please refer to *BasePolicy* for more detailed explanation.

train(*mode: bool = True*) → *ICMPolicy*

Set the module in training mode.

forward(*batch: Batch*, *state: Optional[Union[dict, Batch, ndarray]] = None*, ***kwargs: Any*) → *Batch*

Compute action over the given batch data by inner policy.

See also:

Please refer to *forward()* for more detailed explanation.

exploration_noise(*act: Union[ndarray, Batch]*, *batch: Batch*) → Union[ndarray, *Batch*]

Modify the action from policy.forward with exploration noise.

Parameters

- **act** – a data batch or numpy.ndarray which is the action taken by policy.forward.

- **batch** – the input batch for policy.forward, kept for advanced usage.

Returns
action in the same form of input "act" but with added exploration noise.

set_eps(*eps: float*) → None

Set the eps for epsilon-greedy exploration.

process_fn(*batch:* Batch, *buffer:* ReplayBuffer, *indices: ndarray*) → *Batch*

Pre-process the data from the provided replay buffer.

Used in `update()`. Check out *policy.process_fn* for more information.

post_process_fn(*batch:* Batch, *buffer:* ReplayBuffer, *indices: ndarray*) → None

Post-process the data from the provided replay buffer.

Typical usage is to update the sampling weight in prioritized experience replay. Used in `update()`.

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns
A dict, including the data needed to be logged (e.g., loss).

Note: In order to distinguish the collecting state, updating state and testing state, you can check the policy state by `self.training` and `self.updating`. Please refer to *States for policy* for more detailed explanation.

Warning: If you use `torch.distributions.Normal` and `torch.distributions.Categorical` to calculate the log_prob, please be careful about the shape: Categorical distribution gives "[batch_size]" shape while Normal distribution gives "[batch_size, 1]" shape. The auto-broadcasting of numerical operation with torch tensors will amplify this error.

training: bool

1.11.5 Multi-agent

class tianshou.policy.**MultiAgentPolicyManager**(*policies: List[*BasePolicy*]*, *env:* PettingZooEnv, ***kwargs: Any*)

Bases: *BasePolicy*

Multi-agent policy manager for MARL.

This multi-agent policy manager accepts a list of *BasePolicy*. It dispatches the batch data to each of these policies when the "forward" is called. The same as "process_fn" and "learn": it splits the data and feeds them to each policy. A figure in *Multi-Agent Reinforcement Learning* can help you better understand this procedure.

replace_policy(*policy:* BasePolicy, *agent_id: int*) → None

Replace the "agent_id"th policy in this manager.

process_fn(*batch:* Batch, *buffer:* ReplayBuffer, *indice: ndarray*) → *Batch*

Dispatch batch data from obs.agent_id to every policy's process_fn.

Save original multi-dimensional rew in "save_rew", set rew to the reward of each agent during their "process_fn", and restore the original reward afterwards.

exploration_noise(*act: Union[ndarray,* Batch*]*, *batch:* Batch) → Union[ndarray, *Batch*]

Add exploration noise from sub-policy onto act.

forward(*batch:* Batch, *state: Optional[Union[dict,* Batch*]] = None*, ***kwargs: Any*) → *Batch*

Dispatch batch data from obs.agent_id to every policy's forward.

Parameters

state – if None, it means all agents have no state. If not None, it should contain keys of "agent_1", "agent_2", ...

Returns

a Batch with the following contents:

```
{
    "act": actions corresponding to the input
    "state": {
        "agent_1": output state of agent_1's policy for the state
        "agent_2": xxx
        ...
        "agent_n": xxx}
    "out": {
        "agent_1": output of agent_1's policy for the input
        "agent_2": xxx
        ...
        "agent_n": xxx}
}
```

learn(*batch:* Batch, ***kwargs: Any*) → Dict[str, Union[float, List[float]]]

Dispatch the data to all policies for learning.

Returns

a dict with the following contents:

```
{
    "agent_1/item1": item 1 of agent_1's policy.learn output
    "agent_1/item2": item 2 of agent_1's policy.learn output
    "agent_2/xxx": xxx
    ...
    "agent_n/xxx": xxx
}
```

training: bool

1.12 tianshou.trainer

1.12.1 On-policy

class tianshou.trainer.**OnpolicyTrainer**(*policy: ~tianshou.policy.base.BasePolicy, train_collector: ~tianshou.data.collector.Collector, test_collector: ~typing.Optional[~tianshou.data.collector.Collector], max_epoch: int, step_per_epoch: int, repeat_per_collect: int, episode_per_test: int, batch_size: int, step_per_collect: ~typing.Optional[int] = None, episode_per_collect: ~typing.Optional[int] = None, train_fn: ~typing.Optional[~typing.Callable[[int, int], None]] = None, test_fn: ~typing.Optional[~typing.Callable[[int, ~typing.Optional[int]], None]] = None, stop_fn: ~typing.Optional[~typing.Callable[[float], bool]] = None, save_best_fn: ~typing.Optional[~typing.Callable[[~tianshou.policy.base.BasePolicy], None]] = None, save_checkpoint_fn: ~typing.Optional[~typing.Callable[[int, int, int], str]] = None, resume_from_log: bool = False, reward_metric: ~typing.Optional[~typing.Callable[[~numpy.ndarray], ~numpy.ndarray]] = None, logger: ~tianshou.utils.logger.base.BaseLogger = <tianshou.utils.logger.base.LazyLogger object>, verbose: bool = True, show_progress: bool = True, test_in_train: bool = True, **kwargs: ~typing.Any*)

Bases: BaseTrainer

An iterator class for onpolicy trainer procedure.

Returns an iterator that yields a 3-tuple (epoch, stats, info) of train results on every epoch.

The "step" in onpolicy trainer means an environment step (a.k.a. transition).

Example usage:

```
trainer = OnpolicyTrainer(...)
for epoch, epoch_stat, info in trainer:
    print("Epoch:", epoch)
    print(epoch_stat)
    print(info)
    do_something_with_policy()
    query_something_about_policy()
    make_a_plot_with(epoch_stat)
    display(info)
```

- epoch int: the epoch number
- epoch_stat dict: a large collection of metrics of the current epoch
- info dict: result returned from *gather_info()*

You can even iterate on several trainers at the same time:

```
trainer1 = OnpolicyTrainer(...)
trainer2 = OnpolicyTrainer(...)
for result1, result2, ... in zip(trainer1, trainer2, ...):
    compare_results(result1, result2, ...)
```

Parameters

- **policy** – an instance of the *BasePolicy* class.
- **train_collector** (Collector) – the collector used for training.
- **test_collector** (Collector) – the collector used for testing. If it's None, then no testing will be performed.
- **max_epoch** (*int*) – the maximum number of epochs for training. The training process might be finished before reaching `max_epoch` if `stop_fn` is set.
- **step_per_epoch** (*int*) – the number of transitions collected per epoch.
- **repeat_per_collect** (*int*) – the number of repeat time for policy learning, for example, set it to 2 means the policy needs to learn each given batch data twice.
- **episode_per_test** (*int*) – the number of episodes for one policy evaluation.
- **batch_size** (*int*) – the batch size of sample data, which is going to feed in the policy network.
- **step_per_collect** (*int*) – the number of transitions the collector would collect before the network update, i.e., trainer will collect "step_per_collect" transitions and do some policy network update repeatedly in each epoch.
- **episode_per_collect** (*int*) – the number of episodes the collector would collect before the network update, i.e., trainer will collect "episode_per_collect" episodes and do some policy network update repeatedly in each epoch.
- **train_fn** (*function*) – a hook called at the beginning of training in each epoch. It can be used to perform custom additional operations, with the signature `f(num_epoch: int, step_idx: int) -> None`.
- **test_fn** (*function*) – a hook called at the beginning of testing in each epoch. It can be used to perform custom additional operations, with the signature `f(num_epoch: int, step_idx: int) -> None`.
- **save_best_fn** (*function*) – a hook called when the undiscounted average mean reward in evaluation phase gets better, with the signature `f(policy: BasePolicy) -> None`. It was `save_fn` previously.
- **save_checkpoint_fn** (*function*) – a function to save training process and return the saved checkpoint path, with the signature `f(epoch: int, env_step: int, gradient_step: int) -> str`; you can save whatever you want.
- **resume_from_log** (*bool*) – resume env_step/gradient_step and other metadata from existing tensorboard log. Default to False.
- **stop_fn** (*function*) – a function with signature `f(mean_rewards: float) -> bool`, receives the average undiscounted returns of the testing result, returns a boolean which indicates whether reaching the goal.
- **reward_metric** (*function*) – a function with signature `f(rewards: np.ndarray with shape (num_episode, agent_num)) -> np.ndarray with shape (num_episode,)`, used in multi-agent RL. We need to return a single scalar for each episode's result to monitor training in the multi-agent RL setting. This function specifies what is the desired metric, e.g., the reward of agent 1 or the average reward over all agents.
- **logger** (BaseLogger) – A logger that logs statistics during training/testing/updating. Default to a logger that doesn't log anything.
- **verbose** (*bool*) – whether to print the information. Default to True.
- **show_progress** (*bool*) – whether to display a progress bar when training. Default to True.

- **test_in_train** (*bool*) – whether to test in the training phase. Default to True.

Note: Only either one of step_per_collect and episode_per_collect can be specified.

policy_update_fn(*data: Dict[str, Any]*, *result: Optional[Dict[str, Any]] = None*) → None

Perform one on-policy update.

tianshou.trainer.**onpolicy_trainer**(**args*, ***kwargs*) → Dict[str, Union[float, str]]

Wrapper for OnpolicyTrainer run method.

It is identical to `OnpolicyTrainer(...).run()`.

Returns

See *gather_info()*.

tianshou.trainer.**onpolicy_trainer_iter**

alias of *OnpolicyTrainer*

1.12.2 Off-policy

class tianshou.trainer.**OffpolicyTrainer**(*policy: ~tianshou.policy.base.BasePolicy, train_collector: ~tianshou.data.collector.Collector, test_collector: ~typing.Optional[~tianshou.data.collector.Collector], max_epoch: int, step_per_epoch: int, step_per_collect: int, episode_per_test: int, batch_size: int, update_per_step: ~typing.Union[int, float] = 1, train_fn: ~typing.Optional[~typing.Callable[[int, int], None]] = None, test_fn: ~typing.Optional[~typing.Callable[[int, ~typing.Optional[int]], None]] = None, stop_fn: ~typing.Optional[~typing.Callable[[float], bool]] = None, save_best_fn: ~typing.Optional[~typing.Callable[[~tianshou.policy.base.BasePolicy], None]] = None, save_checkpoint_fn: ~typing.Optional[~typing.Callable[[int, int, int], str]] = None, resume_from_log: bool = False, reward_metric: ~typing.Optional[~typing.Callable[[~numpy.ndarray], ~numpy.ndarray]] = None, logger: ~tianshou.utils.logger.base.BaseLogger = <tianshou.utils.logger.base.LazyLogger object>, verbose: bool = True, show_progress: bool = True, test_in_train: bool = True, **kwargs: ~typing.Any*)

Bases: `BaseTrainer`

An iterator class for offpolicy trainer procedure.

Returns an iterator that yields a 3-tuple (epoch, stats, info) of train results on every epoch.

The "step" in offpolicy trainer means an environment step (a.k.a. transition).

Example usage:

```
trainer = OffpolicyTrainer(...)
for epoch, epoch_stat, info in trainer:
    print("Epoch:", epoch)
```

(continues on next page)

(continued from previous page)

```
    print(epoch_stat)
    print(info)
    do_something_with_policy()
    query_something_about_policy()
    make_a_plot_with(epoch_stat)
    display(info)
```

- epoch int: the epoch number
- epoch_stat dict: a large collection of metrics of the current epoch
- info dict: result returned from *gather_info()*

You can even iterate on several trainers at the same time:

```
trainer1 = OffpolicyTrainer(...)
trainer2 = OffpolicyTrainer(...)
for result1, result2, ... in zip(trainer1, trainer2, ...):
    compare_results(result1, result2, ...)
```

Parameters

- **policy** – an instance of the *BasePolicy* class.
- **train_collector** (Collector) – the collector used for training.
- **test_collector** (Collector) – the collector used for testing. If it's None, then no testing will be performed.
- **max_epoch** (*int*) – the maximum number of epochs for training. The training process might be finished before reaching `max_epoch` if `stop_fn` is set.
- **step_per_epoch** (*int*) – the number of transitions collected per epoch.
- **step_per_collect** (*int*) – the number of transitions the collector would collect before the network update, i.e., trainer will collect "step_per_collect" transitions and do some policy network update repeatedly in each epoch.
- **episode_per_test** – the number of episodes for one policy evaluation.
- **batch_size** (*int*) – the batch size of sample data, which is going to feed in the policy network.
- **update_per_step** (*int/float*) – the number of times the policy network would be updated per transition after (step_per_collect) transitions are collected, e.g., if update_per_step set to 0.3, and step_per_collect is 256 , policy will be updated round(256 * 0.3 = 76.8) = 77 times after 256 transitions are collected by the collector. Default to 1.
- **train_fn** (*function*) – a hook called at the beginning of training in each epoch. It can be used to perform custom additional operations, with the signature `f(num_epoch: int, step_idx: int) -> None`.
- **test_fn** (*function*) – a hook called at the beginning of testing in each epoch. It can be used to perform custom additional operations, with the signature `f(num_epoch: int, step_idx: int) -> None`.
- **save_best_fn** (*function*) – a hook called when the undiscounted average mean reward in evaluation phase gets better, with the signature `f(policy: BasePolicy) -> None`. It was `save_fn` previously.

- **save_checkpoint_fn** (*function*) – a function to save training process and return the saved checkpoint path, with the signature `f(epoch: int, env_step: int, gradient_step: int) -> str`; you can save whatever you want.
- **resume_from_log** (*bool*) – resume env_step/gradient_step and other metadata from existing tensorboard log. Default to False.
- **stop_fn** (*function*) – a function with signature `f(mean_rewards: float) -> bool`, receives the average undiscounted returns of the testing result, returns a boolean which indicates whether reaching the goal.
- **reward_metric** (*function*) – a function with signature `f(rewards: np.ndarray with shape (num_episode, agent_num)) -> np.ndarray with shape (num_episode,)`, used in multi-agent RL. We need to return a single scalar for each episode's result to monitor training in the multi-agent RL setting. This function specifies what is the desired metric, e.g., the reward of agent 1 or the average reward over all agents.
- **logger** (BaseLogger) – A logger that logs statistics during training/testing/updating. Default to a logger that doesn't log anything.
- **verbose** (*bool*) – whether to print the information. Default to True.
- **show_progress** (*bool*) – whether to display a progress bar when training. Default to True.
- **test_in_train** (*bool*) – whether to test in the training phase. Default to True.

policy_update_fn(*data: Dict[str, Any]*, *result: Dict[str, Any]*) → None

Perform off-policy updates.

tianshou.trainer.**offpolicy_trainer**(**args*, ***kwargs*) → Dict[str, Union[float, str]]

Wrapper for OffPolicyTrainer run method.

It is identical to `OffpolicyTrainer(...).run()`.

Returns

See *gather_info()*.

tianshou.trainer.**offpolicy_trainer_iter**

alias of *OffpolicyTrainer*

1.12.3 Offline

class tianshou.trainer.**OfflineTrainer**(*policy: ~tianshou.policy.base.BasePolicy, buffer: ~tianshou.data.buffer.base.ReplayBuffer, test_collector: ~typing.Optional[~tianshou.data.collector.Collector], max_epoch: int, update_per_epoch: int, episode_per_test: int, batch_size: int, test_fn: ~typing.Optional[~typing.Callable[[int, ~typing.Optional[int]], None]] = None, stop_fn: ~typing.Optional[~typing.Callable[[float], bool]] = None, save_best_fn: ~typing.Optional[~typing.Callable[[~tianshou.policy.base.BasePolicy], None]] = None, save_checkpoint_fn: ~typing.Optional[~typing.Callable[[int, int, int], str]] = None, resume_from_log: bool = False, reward_metric: ~typing.Optional[~typing.Callable[[~numpy.ndarray], ~numpy.ndarray]] = None, logger: ~tianshou.utils.logger.base.BaseLogger = <tianshou.utils.logger.base.LazyLogger object>, verbose: bool = True, show_progress: bool = True, **kwargs: ~typing.Any*)

Bases: BaseTrainer

An iterator class for offline trainer procedure.

Returns an iterator that yields a 3-tuple (epoch, stats, info) of train results on every epoch.

The "step" in offline trainer means a gradient step.

Example usage:

```
trainer = OfflineTrainer(...)
for epoch, epoch_stat, info in trainer:
    print("Epoch:", epoch)
    print(epoch_stat)
    print(info)
    do_something_with_policy()
    query_something_about_policy()
    make_a_plot_with(epoch_stat)
    display(info)
```

- epoch int: the epoch number
- epoch_stat dict: a large collection of metrics of the current epoch
- info dict: result returned from *gather_info()*

You can even iterate on several trainers at the same time:

```
trainer1 = OfflineTrainer(...)
trainer2 = OfflineTrainer(...)
for result1, result2, ... in zip(trainer1, trainer2, ...):
    compare_results(result1, result2, ...)
```

Parameters

- **policy** – an instance of the *BasePolicy* class.
- **buffer** – an instance of the *ReplayBuffer* class. This buffer must be populated with experiences for offline RL.
- **test_collector** (Collector) – the collector used for testing. If it's None, then no testing will be performed.
- **max_epoch** (*int*) – the maximum number of epochs for training. The training process might be finished before reaching max_epoch if stop_fn is set.
- **update_per_epoch** (*int*) – the number of policy network updates, so-called gradient steps, per epoch.
- **episode_per_test** – the number of episodes for one policy evaluation.
- **batch_size** (*int*) – the batch size of sample data, which is going to feed in the policy network.
- **test_fn** (*function*) – a hook called at the beginning of testing in each epoch. It can be used to perform custom additional operations, with the signature f(num_epoch: int, step_idx: int) -> None.

- **save_best_fn** (*function*) – a hook called when the undiscounted average mean reward in evaluation phase gets better, with the signature `f(policy: BasePolicy) -> None`. It was `save_fn` previously.
- **save_checkpoint_fn** (*function*) – a function to save training process and return the saved checkpoint path, with the signature `f(epoch: int, env_step: int, gradient_step: int) -> str`; you can save whatever you want. Because offline-RL doesn't have env_step, the env_step is always 0 here.
- **resume_from_log** (*bool*) – resume gradient_step and other metadata from existing tensorboard log. Default to False.
- **stop_fn** (*function*) – a function with signature `f(mean_rewards: float) -> bool`, receives the average undiscounted returns of the testing result, returns a boolean which indicates whether reaching the goal.
- **reward_metric** (*function*) – a function with signature `f(rewards: np.ndarray with shape (num_episode, agent_num)) -> np.ndarray with shape (num_episode,)`, used in multi-agent RL. We need to return a single scalar for each episode's result to monitor training in the multi-agent RL setting. This function specifies what is the desired metric, e.g., the reward of agent 1 or the average reward over all agents.
- **logger** (BaseLogger) – A logger that logs statistics during updating/testing. Default to a logger that doesn't log anything.
- **verbose** (*bool*) – whether to print the information. Default to True.
- **show_progress** (*bool*) – whether to display a progress bar when training. Default to True.

policy_update_fn(*data: Dict[str, Any]*, *result: Optional[Dict[str, Any]] = None*) → None

Perform one off-line policy update.

tianshou.trainer.**offline_trainer**(**args*, ***kwargs*) → Dict[str, Union[float, str]]

Wrapper for offline_trainer run method.

It is identical to `OfflineTrainer(...).run()`.

Returns

See *gather_info()*.

tianshou.trainer.**offline_trainer_iter**

alias of *OfflineTrainer*

1.12.4 utils

tianshou.trainer.**test_episode**(*policy: BasePolicy*, *collector: Collector*, *test_fn: Optional[Callable[[int, Optional[int]], None]]*, *epoch: int*, *n_episode: int*, *logger: Optional[BaseLogger] = None*, *global_step: Optional[int] = None*, *reward_metric: Optional[Callable[[ndarray], ndarray]] = None*) → Dict[str, Any]

A simple wrapper of testing policy in collector.

tianshou.trainer.**gather_info**(*start_time: float*, *train_collector: Optional[Collector]*, *test_collector: Optional[Collector]*, *best_reward: float*, *best_reward_std: float*) → Dict[str, Union[float, str]]

A simple wrapper of gathering information from collectors.

Returns

A dictionary with the following keys:

- `train_step` the total collected step of training collector;
- `train_episode` the total collected episode of training collector;
- `train_time/collector` the time for collecting transitions in the training collector;
- `train_time/model` the time for training models;
- `train_speed` the speed of training (env_step per second);
- `test_step` the total collected step of test collector;
- `test_episode` the total collected episode of test collector;
- `test_time` the time for testing;
- `test_speed` the speed of testing (env_step per second);
- `best_reward` the best reward over the test results;
- `duration` the total elapsed time.

1.13 tianshou.exploration

class tianshou.exploration.**BaseNoise**

Bases: ABC, object

The action noise base class.

reset() → None

Reset to the initial state.

abstract __call__(*size: Sequence[int]*) → ndarray

Generate new noise.

class tianshou.exploration.**GaussianNoise**(*mu: float = 0.0*, *sigma: float = 1.0*)

Bases: *BaseNoise*

The vanilla Gaussian process, for exploration in DDPG by default.

__call__(*size: Sequence[int]*) → ndarray

Generate new noise.

class tianshou.exploration.**OUNoise**(*mu: float = 0.0*, *sigma: float = 0.3*, *theta: float = 0.15*, *dt: float = 0.01*, *x0: Optional[Union[float, ndarray]] = None*)

Bases: *BaseNoise*

Class for Ornstein-Uhlenbeck process, as used for exploration in DDPG.

Usage:

```
# init
self.noise = OUNoise()
# generate noise
noise = self.noise(logits.shape, eps)
```

For required parameters, you can refer to the stackoverflow page. However, our experiment result shows that (similar to OpenAI SpinningUp) using vanilla Gaussian process has little difference from using the Ornstein-Uhlenbeck process.

reset() → None

Reset to the initial state.

__call__(*size: Sequence[int]*, *mu: Optional[float] = None*) → ndarray

Generate new noise.

Return an numpy array which size is equal to `size`.

1.14 tianshou.utils

Utils package.

class tianshou.utils.**MovAvg**(*size: int = 100*)

Bases: `object`

Class for moving average.

It will automatically exclude the infinity and NaN. Usage:

```
>>> stat = MovAvg(size=66)
>>> stat.add(torch.tensor(5))
5.0
>>> stat.add(float('inf'))  # which will not add to stat
5.0
>>> stat.add([6, 7, 8])
6.5
>>> stat.get()
6.5
>>> print(f'{stat.mean():.2f}±{stat.std():.2f}')
6.50±1.12
```

add(*data_array: Union[Number, number, list, ndarray, Tensor]*) → float

Add a scalar into *MovAvg*.

You can add `torch.Tensor` with only one element, a python scalar, or a list of python scalar.

get() → float

Get the average.

mean() → float

Get the average. Same as *get()*.

std() → float

Get the standard deviation.

class tianshou.utils.**RunningMeanStd**(*mean: Union[float, ndarray] = 0.0*, *std: Union[float, ndarray] = 1.0*, *clip_max: Optional[float] = 10.0*, *epsilon: float = 1.1920928955078125e-07*)

Bases: `object`

Calculates the running mean and std of a data stream.

https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Parallel_algorithm

Parameters

- **mean** – the initial mean estimation for data array. Default to 0.
- **std** – the initial standard error estimation for data array. Default to 1.
- **clip_max** (*float*) – the maximum absolute value for data array. Default to 10.0.
- **epsilon** (*float*) – To avoid division by zero.

norm(*data_array: Union[float, ndarray]*) → Union[float, ndarray]

update(*data_array: ndarray*) → None

Add a batch of item into RMS with the same shape, modify mean/var/count.

class tianshou.utils.**DummyTqdm**(*total: int*, ***kwargs: Any*)

Bases: object

A dummy tqdm class that keeps stats but without progress bar.

It supports __enter__ and __exit__, update and a dummy set_postfix, which is the interface that trainers use.

Note: Using disable=True in tqdm config results in infinite loop, thus this class is created. See the discussion at #641 for details.

---

set_postfix(***kwargs: Any*) → None

update(*n: int = 1*) → None

class tianshou.utils.**BaseLogger**(*train_interval: int = 1000*, *test_interval: int = 1*, *update_interval: int = 1000*)

Bases: ABC

The base class for any logger which is compatible with trainer.

Try to overwrite write() method to use your own writer.

Parameters

- **train_interval** (*int*) – the log interval in log_train_data(). Default to 1000.
- **test_interval** (*int*) – the log interval in log_test_data(). Default to 1.
- **update_interval** (*int*) – the log interval in log_update_data(). Default to 1000.

abstract write(*step_type: str*, *step: int*, *data: Dict[str, Union[int, Number, number, ndarray]]*) → None

Specify how the writer is used to log data.

Parameters

- **step_type** (*str*) – namespace which the data dict belongs to.
- **step** (*int*) – stands for the ordinate of the data dict.
- **data** (*dict*) – the data to write with format {key: value}.

log_train_data(*collect_result: dict*, *step: int*) → None

Use writer to log statistics generated during training.

Parameters

- **collect_result** – a dict containing information of data collected in training stage, i.e., returns of collector.collect().

- **step** (*int*) – stands for the timestep the collect_result being logged.

log_test_data(*collect_result: dict*, *step: int*) → None

Use writer to log statistics generated during evaluating.

Parameters

- **collect_result** – a dict containing information of data collected in evaluating stage, i.e., returns of collector.collect().
- **step** (*int*) – stands for the timestep the collect_result being logged.

log_update_data(*update_result: dict*, *step: int*) → None

Use writer to log statistics generated during updating.

Parameters

- **update_result** – a dict containing information of data collected in updating stage, i.e., returns of policy.update().
- **step** (*int*) – stands for the timestep the collect_result being logged.

abstract save_data(*epoch: int*, *env_step: int*, *gradient_step: int*, *save_checkpoint_fn: Optional[Callable[[int, int, int], str]] = None*) → None

Use writer to log metadata when calling `save_checkpoint_fn` in trainer.

Parameters

- **epoch** (*int*) – the epoch in trainer.
- **env_step** (*int*) – the env_step in trainer.
- **gradient_step** (*int*) – the gradient_step in trainer.
- **save_checkpoint_fn** (*function*) – a hook defined by user, see trainer documentation for detail.

abstract restore_data() → Tuple[int, int, int]

Return the metadata from existing log.

If it finds nothing or an error occurs during the recover process, it will return the default parameters.

Returns

epoch, env_step, gradient_step.

class tianshou.utils.**TensorboardLogger**(*writer: SummaryWriter*, *train_interval: int = 1000*, *test_interval: int = 1*, *update_interval: int = 1000*, *save_interval: int = 1*, *write_flush: bool = True*)

Bases: *BaseLogger*

A logger that relies on tensorboard SummaryWriter by default to visualize and log statistics.

Parameters

- **writer** (*SummaryWriter*) – the writer to log data.
- **train_interval** (*int*) – the log interval in log_train_data(). Default to 1000.
- **test_interval** (*int*) – the log interval in log_test_data(). Default to 1.
- **update_interval** (*int*) – the log interval in log_update_data(). Default to 1000.
- **save_interval** (*int*) – the save interval in save_data(). Default to 1 (save at the end of each epoch).

- **write_flush** (*bool*) – whether to flush tensorboard result after each add_scalar operation. Default to True.

write(*step_type: str*, *step: int*, *data: Dict[str, Union[int, Number, number, ndarray]]*) → None

Specify how the writer is used to log data.

Parameters

- **step_type** (*str*) – namespace which the data dict belongs to.
- **step** (*int*) – stands for the ordinate of the data dict.
- **data** (*dict*) – the data to write with format `{key: value}`.

save_data(*epoch: int*, *env_step: int*, *gradient_step: int*, *save_checkpoint_fn: Optional[Callable[[int, int, int], str]] = None*) → None

Use writer to log metadata when calling `save_checkpoint_fn` in trainer.

Parameters

- **epoch** (*int*) – the epoch in trainer.
- **env_step** (*int*) – the env_step in trainer.
- **gradient_step** (*int*) – the gradient_step in trainer.
- **save_checkpoint_fn** (*function*) – a hook defined by user, see trainer documentation for detail.

restore_data() → Tuple[int, int, int]

Return the metadata from existing log.

If it finds nothing or an error occurs during the recover process, it will return the default parameters.

Returns
epoch, env_step, gradient_step.

class tianshou.utils.**BasicLogger**(**args: Any*, ***kwargs: Any*)

Bases: *TensorboardLogger*

BasicLogger has changed its name to TensorboardLogger in #427.

This class is for compatibility.

class tianshou.utils.**LazyLogger**

Bases: *BaseLogger*

A logger that does nothing. Used as the placeholder in trainer.

write(*step_type: str*, *step: int*, *data: Dict[str, Union[int, Number, number, ndarray]]*) → None

The LazyLogger writes nothing.

save_data(*epoch: int*, *env_step: int*, *gradient_step: int*, *save_checkpoint_fn: Optional[Callable[[int, int, int], str]] = None*) → None

Use writer to log metadata when calling `save_checkpoint_fn` in trainer.

Parameters

- **epoch** (*int*) – the epoch in trainer.
- **env_step** (*int*) – the env_step in trainer.
- **gradient_step** (*int*) – the gradient_step in trainer.

- **save_checkpoint_fn** (*function*) – a hook defined by user, see trainer documentation for detail.

restore_data() → Tuple[int, int, int]

Return the metadata from existing log.

If it finds nothing or an error occurs during the recover process, it will return the default parameters.

Returns

epoch, env_step, gradient_step.

class tianshou.utils.**WandbLogger**(*train_interval: int = 1000*, *test_interval: int = 1*, *update_interval: int = 1000*, *save_interval: int = 1000*, *write_flush: bool = True*, *project: Optional[str] = None*, *name: Optional[str] = None*, *entity: Optional[str] = None*, *run_id: Optional[str] = None*, *config: Optional[Namespace] = None*)

Bases: *BaseLogger*

Weights and Biases logger that sends data to https://wandb.ai/.

This logger creates three panels with plots: train, test, and update. Make sure to select the correct access for each panel in weights and biases:

Example of usage:

```
logger = WandbLogger()
logger.load(SummaryWriter(log_path))
result = onpolicy_trainer(policy, train_collector, test_collector,
                          logger=logger)
```

Parameters

- **train_interval** (*int*) – the log interval in log_train_data(). Default to 1000.
- **test_interval** (*int*) – the log interval in log_test_data(). Default to 1.
- **update_interval** (*int*) – the log interval in log_update_data(). Default to 1000.
- **save_interval** (*int*) – the save interval in save_data(). Default to 1 (save at the end of each epoch).
- **write_flush** (*bool*) – whether to flush tensorboard result after each add_scalar operation. Default to True.
- **project** (*str*) – W&B project name. Default to "tianshou".
- **name** (*str*) – W&B run name. Default to None. If None, random name is assigned.
- **entity** (*str*) – W&B team/organization name. Default to None.
- **run_id** (*str*) – run id of W&B run to be resumed. Default to None.
- **config** (*argparse.Namespace*) – experiment configurations. Default to None.

load(*writer: SummaryWriter*) → None

write(*step_type: str*, *step: int*, *data: Dict[str, Union[int, Number, number, ndarray]]*) → None

Specify how the writer is used to log data.

Parameters

- **step_type** (*str*) – namespace which the data dict belongs to.

- **step** (*int*) – stands for the ordinate of the data dict.
- **data** (*dict*) – the data to write with format `{key: value}`.

save_data(*epoch: int*, *env_step: int*, *gradient_step: int*, *save_checkpoint_fn: Optional[Callable[[int, int, int], str]] = None*) → None

Use writer to log metadata when calling `save_checkpoint_fn` in trainer.

Parameters

- **epoch** (*int*) – the epoch in trainer.
- **env_step** (*int*) – the env_step in trainer.
- **gradient_step** (*int*) – the gradient_step in trainer.
- **save_checkpoint_fn** (*function*) – a hook defined by user, see trainer documentation for detail.

restore_data() → Tuple[int, int, int]

Return the metadata from existing log.

If it finds nothing or an error occurs during the recover process, it will return the default parameters.

Returns
epoch, env_step, gradient_step.

tianshou.utils.**deprecation**(*msg: str*) → None

Deprecation warning wrapper.

class tianshou.utils.**MultipleLRSchedulers**(**args: LambdaLR*)

Bases: `object`

A wrapper for multiple learning rate schedulers.

Every time *step()* is called, it calls the step() method of each of the schedulers that it contains. Example usage:

```
scheduler1 = ConstantLR(opt1, factor=0.1, total_iters=2)
scheduler2 = ExponentialLR(opt2, gamma=0.9)
scheduler = MultipleLRSchedulers(scheduler1, scheduler2)
policy = PPOPolicy(..., lr_scheduler=scheduler)
```

step() → None

Take a step in each of the learning rate schedulers.

state_dict() → List[Dict]

Get state_dict for each of the learning rate schedulers.

Returns
A list of state_dict of learning rate schedulers.

load_state_dict(*state_dict: List[Dict]*) → None

Load states from state_dict.

Parameters
state_dict (*List[Dict]*) – A list of learning rate scheduler state_dict, in the same order as the schedulers.

1.14.1 Pre-defined Networks

Common

tianshou.utils.net.common.**miniblock**(*input_size: int, output_size: int = 0, norm_layer: ~typing.Optional[~typing.Type[~torch.nn.modules.module.Module]] = None, norm_args: ~typing.Optional[~typing.Union[~typing.Tuple[~typing.Any, ...], ~typing.Dict[~typing.Any, ~typing.Any]]] = None, activation: ~typing.Optional[~typing.Type[~torch.nn.modules.module.Module]] = None, act_args: ~typing.Optional[~typing.Union[~typing.Tuple[~typing.Any, ...], ~typing.Dict[~typing.Any, ~typing.Any]]] = None, linear_layer: ~typing.Type[~torch.nn.modules.linear.Linear] = <class 'torch.nn.modules.linear.Linear'>*) → List[Module]

Construct a miniblock with given input/output-size, norm layer and activation.

class tianshou.utils.net.common.**MLP**(*input_dim: int, output_dim: int = 0, hidden_sizes: ~typing.Sequence[int] = (), norm_layer: ~typing.Optional[~typing.Union[~typing.Type[~torch.nn.modules.module.Module], ~typing.Sequence[~typing.Type[~torch.nn.modules.module.Module]]]] = None, norm_args: ~typing.Optional[~typing.Union[~typing.Tuple[~typing.Any, ...], ~typing.Dict[~typing.Any, ~typing.Any], ~typing.Sequence[~typing.Tuple[~typing.Any, ...]], ~typing.Sequence[~typing.Dict[~typing.Any, ~typing.Any]]]] = None, activation: ~typing.Optional[~typing.Union[~typing.Type[~torch.nn.modules.module.Module], ~typing.Sequence[~typing.Type[~torch.nn.modules.module.Module]]]] = <class 'torch.nn.modules.activation.ReLU'>, act_args: ~typing.Optional[~typing.Union[~typing.Tuple[~typing.Any, ...], ~typing.Dict[~typing.Any, ~typing.Any], ~typing.Sequence[~typing.Tuple[~typing.Any, ...]], ~typing.Sequence[~typing.Dict[~typing.Any, ~typing.Any]]]] = None, device: ~typing.Optional[~typing.Union[str, int, ~torch.device]] = None, linear_layer: ~typing.Type[~torch.nn.modules.linear.Linear] = <class 'torch.nn.modules.linear.Linear'>, flatten_input: bool = True*)

Bases: `Module`

Simple MLP backbone.

Create a MLP of size input_dim * hidden_sizes[0] * hidden_sizes[1] * ... * hidden_sizes[-1] * output_dim

Parameters

- **input_dim** (*int*) – dimension of the input vector.
- **output_dim** (*int*) – dimension of the output vector. If set to 0, there is no final linear layer.
- **hidden_sizes** – shape of MLP passed in as a list, not including input_dim and output_dim.
- **norm_layer** – use which normalization before activation, e.g., `nn.LayerNorm` and `nn.BatchNorm1d`. Default to no normalization. You can also pass a list of normalization modules with the same length of hidden_sizes, to use different normalization module in different layers. Default to no normalization.

- **activation** – which activation to use after each layer, can be both the same activation for all layers if passed in nn.Module, or different activation for different Modules if passed in a list. Default to nn.ReLU.
- **device** – which device to create this model on. Default to None.
- **linear_layer** – use this module as linear layer. Default to nn.Linear.
- **flatten_input** (*bool*) – whether to flatten input data. Default to True.

forward(*obs: Union[ndarray, Tensor]*) → Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class tianshou.utils.net.common.**Net**(*state_shape: ~typing.Union[int, ~typing.Sequence[int]], action_shape: ~typing.Union[int, ~typing.Sequence[int]] = 0, hidden_sizes: ~typing.Sequence[int] = (), norm_layer: ~typing.Optional[~typing.Union[~typing.Type[~torch.nn.modules.module.Module], ~typing.Sequence[~typing.Type[~torch.nn.modules.module.Module]]]] = None, norm_args: ~typing.Optional[~typing.Union[~typing.Tuple[~typing.Any, ...], ~typing.Dict[~typing.Any, ~typing.Any], ~typing.Sequence[~typing.Tuple[~typing.Any, ...]], ~typing.Sequence[~typing.Dict[~typing.Any, ~typing.Any]]]] = None, activation: ~typing.Optional[~typing.Union[~typing.Type[~torch.nn.modules.module.Module], ~typing.Sequence[~typing.Type[~torch.nn.modules.module.Module]]]] = <class 'torch.nn.modules.activation.ReLU'>, act_args: ~typing.Optional[~typing.Union[~typing.Tuple[~typing.Any, ...], ~typing.Dict[~typing.Any, ~typing.Any], ~typing.Sequence[~typing.Tuple[~typing.Any, ...]], ~typing.Sequence[~typing.Dict[~typing.Any, ~typing.Any]]]] = None, device: ~typing.Union[str, int, ~torch.device] = 'cpu', softmax: bool = False, concat: bool = False, num_atoms: int = 1, dueling_param: ~typing.Optional[~typing.Tuple[~typing.Dict[str, ~typing.Any], ~typing.Dict[str, ~typing.Any]]] = None, linear_layer: ~typing.Type[~torch.nn.modules.linear.Linear] = <class 'torch.nn.modules.linear.Linear'>*)

Bases: `Module`

Wrapper of MLP to support more specific DRL usage.

For advanced usage (how to customize the network), please refer to *Build the Network*.

Parameters

- **state_shape** – int or a sequence of int of the shape of state.
- **action_shape** – int or a sequence of int of the shape of action.

- **hidden_sizes** – shape of MLP passed in as a list.
- **norm_layer** – use which normalization before activation, e.g., `nn.LayerNorm` and `nn.BatchNorm1d`. Default to no normalization. You can also pass a list of normalization modules with the same length of hidden_sizes, to use different normalization module in different layers. Default to no normalization.
- **activation** – which activation to use after each layer, can be both the same activation for all layers if passed in nn.Module, or different activation for different Modules if passed in a list. Default to nn.ReLU.
- **device** – specify the device when the network actually runs. Default to "cpu".
- **softmax** (*bool*) – whether to apply a softmax layer over the last layer's output.
- **concat** (*bool*) – whether the input shape is concatenated by state_shape and action_shape. If it is True, `action_shape` is not the output shape, but affects the input shape only.
- **num_atoms** (*int*) – in order to expand to the net of distributional RL. Default to 1 (not use).
- **dueling_param** (*bool*) – whether to use dueling network to calculate Q values (for Dueling DQN). If you want to use dueling option, you should pass a tuple of two dict (first for Q and second for V) stating self-defined arguments as stated in class:*~tianshou.utils.net.common.MLP*. Default to None.
- **linear_layer** – use this module as linear layer. Default to nn.Linear.

See also:

Please refer to *MLP* for more detailed explanation on the usage of activation, norm_layer, etc.

You can also refer to *Actor*, *Critic*, etc, to see how it's suggested be used.

forward(*obs: Union[ndarray, Tensor]*, *state: Optional[Any] = None*, *info: Dict[str, Any] = {}*) → Tuple[Tensor, Any]

Mapping: obs -> flatten (inside MLP)-> logits.

training: bool

class tianshou.utils.net.common.**Recurrent**(*layer_num: int*, *state_shape: Union[int, Sequence[int]]*, *action_shape: Union[int, Sequence[int]]*, *device: Union[str, int, device] = 'cpu'*, *hidden_layer_size: int = 128*)

Bases: `Module`

Simple Recurrent network based on LSTM.

For advanced usage (how to customize the network), please refer to *Build the Network*.

forward(*obs: Union[ndarray, Tensor]*, *state: Optional[Dict[str, Tensor]] = None*, *info: Dict[str, Any] = {}*) → Tuple[Tensor, Dict[str, Tensor]]

Mapping: obs -> flatten -> logits.

In the evaluation mode, *obs* should be with shape `[bsz, dim]`; in the training mode, *obs* should be with shape `[bsz, len, dim]`. See the code and comment for more detail.

training: bool

class tianshou.utils.net.common.**ActorCritic**(*actor: Module*, *critic: Module*)

Bases: `Module`

An actor-critic network for parsing parameters.

Using `actor_critic.parameters()` instead of set.union or list+list to avoid issue #449.

Parameters

- **actor** (*nn.Module*) – the actor network.
- **critic** (*nn.Module*) – the critic network.

training: bool

class tianshou.utils.net.common.**DataParallelNet**(*net: Module*)

Bases: Module

DataParallel wrapper for training agent with multi-GPU.

This class does only the conversion of input data type, from numpy array to torch's Tensor. If the input is a nested dictionary, the user should create a similar class to do the same thing.

Parameters

net (*nn.Module*) – the network to be distributed in different GPUs.

forward(*obs: Union[ndarray, Tensor]*, **args: Any*, ***kwargs: Any*) → Tuple[Any, Any]

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class tianshou.utils.net.common.**EnsembleLinear**(*ensemble_size: int*, *in_feature: int*, *out_feature: int*, *bias: bool = True*)

Bases: Module

Linear Layer of Ensemble network.

Parameters

- **ensemble_size** (*int*) – Number of subnets in the ensemble.
- **inp_feature** (*int*) – dimension of the input vector.
- **out_feature** (*int*) – dimension of the output vector.
- **bias** (*bool*) – whether to include an additive bias, default to be True.

forward(*x: Tensor*) → Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class tianshou.utils.net.common.**BranchingNet**(*state_shape: ~typing.Union[int, ~typing.Sequence[int]], num_branches: int = 0, action_per_branch: int = 2, common_hidden_sizes: ~typing.List[int] = [], value_hidden_sizes: ~typing.List[int] = [], action_hidden_sizes: ~typing.List[int] = [], norm_layer: ~typing.Optional[~typing.Type[~torch.nn.modules.module.Module]] = None, norm_args: ~typing.Optional[~typing.Union[~typing.Tuple[~typing.Any, ...], ~typing.Dict[~typing.Any, ~typing.Any], ~typing.Sequence[~typing.Tuple[~typing.Any, ...]], ~typing.Sequence[~typing.Dict[~typing.Any, ~typing.Any]]]] = None, activation: ~typing.Optional[~typing.Type[~torch.nn.modules.module.Module]] = <class 'torch.nn.modules.activation.ReLU'>, act_args: ~typing.Optional[~typing.Union[~typing.Tuple[~typing.Any, ...], ~typing.Dict[~typing.Any, ~typing.Any], ~typing.Sequence[~typing.Tuple[~typing.Any, ...]], ~typing.Sequence[~typing.Dict[~typing.Any, ~typing.Any]]]] = None, device: ~typing.Union[str, int, ~torch.device] = 'cpu'*)

Bases: `Module`

Branching dual Q network.

Network for the BranchingDQNPolicy, it uses a common network module, a value module and action "branches" one for each dimension.It allows for a linear scaling of Q-value the output w.r.t. the number of dimensions in the action space. For more info please refer to: arXiv:1711.08946. :param state_shape: int or a sequence of int of the shape of state. :param action_shape: int or a sequence of int of the shape of action. :param action_peer_branch: int or a sequence of int of the number of actions in each dimension. :param common_hidden_sizes: shape of the common MLP network passed in as a list. :param value_hidden_sizes: shape of the value MLP network passed in as a list. :param action_hidden_sizes: shape of the action MLP network passed in as a list. :param norm_layer: use which normalization before activation, e.g., `nn.LayerNorm` and `nn.BatchNorm1d`. Default to no normalization. You can also pass a list of normalization modules with the same length of hidden_sizes, to use different normalization module in different layers. Default to no normalization. :param activation: which activation to use after each layer, can be both the same activation for all layers if passed in nn.Module, or different activation for different Modules if passed in a list. Default to nn.ReLU. :param device: specify the device when the network actually runs. Default to "cpu". :param bool softmax: whether to apply a softmax layer over the last layer's output.

training: bool

forward(*obs: Union[ndarray, Tensor]*, *state: Optional[Any] = None*, *info: Dict[str, Any] = {}*) → Tuple[Tensor, Any]

Mapping: obs -> model -> logits.

tianshou.utils.net.common.**get_dict_state_decorator**(*state_shape: Dict[str, Union[int, Sequence[int]]]*, *keys: Sequence[str]*) → Tuple[Callable, int]

A helper function to make Net or equivalent classes (e.g. Actor, Critic) applicable to dict state.

The first return item, `decorator_fn`, will alter the implementation of forward function of the given class by preprocessing the observation. The preprocessing is basically flatten the observation and concatenate them based on the `keys` order. The batch dimension is preserved if presented. The result observation shape will be equal to `new_state_shape`, the second return item.

Parameters

- **state_shape** – A dictionary indicating each state's shape
- **keys** – A list of state's keys. The flatten observation will be according to this list order.

Returns

a 2-items tuple `decorator_fn` and `new_state_shape`

Discrete

class tianshou.utils.net.discrete.**Actor**(*preprocess_net: Module*, *action_shape: Sequence[int]*, *hidden_sizes: Sequence[int] = ()*, *softmax_output: bool = True*, *preprocess_net_output_dim: Optional[int] = None*, *device: Union[str, int, device] = 'cpu'*)

Bases: `Module`

Simple actor network.

Will create an actor operated in discrete action space with structure of preprocess_net —> action_shape.

Parameters

- **preprocess_net** – a self-defined preprocess_net which output a flattened hidden state.
- **action_shape** – a sequence of int for the shape of action.
- **hidden_sizes** – a sequence of int for constructing the MLP after preprocess_net. Default to empty sequence (where the MLP now contains only a single linear layer).
- **softmax_output** (*bool*) – whether to apply a softmax layer over the last layer's output.
- **preprocess_net_output_dim** (*int*) – the output dimension of preprocess_net.

For advanced usage (how to customize the network), please refer to *Build the Network*.

See also:

Please refer to `Net` as an instance of how preprocess_net is suggested to be defined.

forward(*obs: Union[ndarray, Tensor]*, *state: Optional[Any] = None*, *info: Dict[str, Any] = {}*) → Tuple[Tensor, Any]

Mapping: s -> Q(s, *).

training: bool

class tianshou.utils.net.discrete.**Critic**(*preprocess_net: Module*, *hidden_sizes: Sequence[int] = ()*, *last_size: int = 1*, *preprocess_net_output_dim: Optional[int] = None*, *device: Union[str, int, device] = 'cpu'*)

Bases: `Module`

Simple critic network. Will create an actor operated in discrete action space with structure of preprocess_net —> 1(q value).

Parameters

- **preprocess_net** – a self-defined preprocess_net which output a flattened hidden state.
- **hidden_sizes** – a sequence of int for constructing the MLP after preprocess_net. Default to empty sequence (where the MLP now contains only a single linear layer).
- **last_size** (*int*) – the output dimension of Critic network. Default to 1.
- **preprocess_net_output_dim** (*int*) – the output dimension of preprocess_net.

For advanced usage (how to customize the network), please refer to *Build the Network*.

See also:

Please refer to `Net` as an instance of how preprocess_net is suggested to be defined.

forward(*obs: Union[ndarray, Tensor]*, ***kwargs: Any*) → Tensor

Mapping: s -> V(s).

training: bool

class tianshou.utils.net.discrete.**CosineEmbeddingNetwork**(*num_cosines: int*, *embedding_dim: int*)

Bases: `Module`

Cosine embedding network for IQN. Convert a scalar in [0, 1] to a list of n-dim vectors.

Parameters

- **num_cosines** – the number of cosines used for the embedding.
- **embedding_dim** – the dimension of the embedding/output.

Note: From https://github.com/ku2482/fqf-iqn-qrdqn.pytorch/blob/master /fqf_iqn_qrdqn/network.py .

---

forward(*taus: Tensor*) → Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class tianshou.utils.net.discrete.**ImplicitQuantileNetwork**(*preprocess_net: Module*, *action_shape: Sequence[int]*, *hidden_sizes: Sequence[int] = ()*, *num_cosines: int = 64*, *preprocess_net_output_dim: Optional[int] = None*, *device: Union[str, int, device] = 'cpu'*)

Bases: *Critic*

Implicit Quantile Network.

Parameters

- **preprocess_net** – a self-defined preprocess_net which output a flattened hidden state.
- **action_shape** (*int*) – a sequence of int for the shape of action.
- **hidden_sizes** – a sequence of int for constructing the MLP after preprocess_net. Default to empty sequence (where the MLP now contains only a single linear layer).
- **num_cosines** (*int*) – the number of cosines to use for cosine embedding. Default to 64.
- **preprocess_net_output_dim** (*int*) – the output dimension of preprocess_net.

Note: Although this class inherits Critic, it is actually a quantile Q-Network with output shape (batch_size, action_dim, sample_size).

The second item of the first return value is tau vector.

forward(*obs: Union[ndarray, Tensor]*, *sample_size: int*, ***kwargs: Any*) → Tuple[Any, Tensor]

Mapping: s -> Q(s, *).

training: bool

class tianshou.utils.net.discrete.**FractionProposalNetwork**(*num_fractions: int*, *embedding_dim: int*)

Bases: `Module`

Fraction proposal network for FQF.

Parameters

- **num_fractions** – the number of factions to propose.
- **embedding_dim** – the dimension of the embedding/input.

Note: Adapted from https://github.com/ku2482/fqf-iqn-qrdqn.pytorch/blob/master /fqf_iqn_qrdqn/network.py .

forward(*obs_embeddings: Tensor*) → Tuple[Tensor, Tensor, Tensor]

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class tianshou.utils.net.discrete.**FullQuantileFunction**(*preprocess_net: Module*, *action_shape: Sequence[int]*, *hidden_sizes: Sequence[int] = ()*, *num_cosines: int = 64*, *preprocess_net_output_dim: Optional[int] = None*, *device: Union[str, int, device] = 'cpu'*)

Bases: *ImplicitQuantileNetwork*

Full(y parameterized) Quantile Function.

Parameters

- **preprocess_net** – a self-defined preprocess_net which output a flattened hidden state.
- **action_shape** (*int*) – a sequence of int for the shape of action.
- **hidden_sizes** – a sequence of int for constructing the MLP after preprocess_net. Default to empty sequence (where the MLP now contains only a single linear layer).
- **num_cosines** (*int*) – the number of cosines to use for cosine embedding. Default to 64.

- **preprocess_net_output_dim** (*int*) – the output dimension of preprocess_net.

Note: The first return value is a tuple of (quantiles, fractions, quantiles_tau), where fractions is a Batch(taus, tau_hats, entropies).

---

forward(*obs: Union[ndarray, Tensor]*, *propose_model:* FractionProposalNetwork, *fractions: Optional[*Batch*] = None*, ***kwargs: Any*) → Tuple[Any, Tensor]

Mapping: s -> Q(s, *).

training: bool

class tianshou.utils.net.discrete.**NoisyLinear**(*in_features: int*, *out_features: int*, *noisy_std: float = 0.5*)

Bases: Module

Implementation of Noisy Networks. arXiv:1706.10295.

Parameters

- **in_features** (*int*) – the number of input features.
- **out_features** (*int*) – the number of output features.
- **noisy_std** (*float*) – initial standard deviation of noisy linear layers.

Note: Adapted from https://github.com/ku2482/fqf-iqn-qrdqn.pytorch/blob/master /fqf_iqn_qrdqn/network.py .

---

reset() → None

f(*x: Tensor*) → Tensor

sample() → None

forward(*x: Tensor*) → Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

tianshou.utils.net.discrete.**sample_noise**(*model: Module*) → bool

Sample the random noises of NoisyLinear modules in the model.

Parameters

model – a PyTorch module which may have NoisyLinear submodules.

Returns

True if model has at least one NoisyLinear submodule; otherwise, False.

class tianshou.utils.net.discrete.**IntrinsicCuriosityModule**(*feature_net: Module*, *feature_dim: int*, *action_dim: int*, *hidden_sizes: Sequence[int] = ()*, *device: Union[str, device] = 'cpu'*)

Bases: Module

Implementation of Intrinsic Curiosity Module. arXiv:1705.05363.

Parameters

- **feature_net** (*torch.nn.Module*) – a self-defined feature_net which output a flattened hidden state.
- **feature_dim** (*int*) – input dimension of the feature net.
- **action_dim** (*int*) – dimension of the action space.
- **hidden_sizes** – hidden layer sizes for forward and inverse models.
- **device** – device for the module.

forward(*s1: Union[ndarray, Tensor]*, *act: Union[ndarray, Tensor]*, *s2: Union[ndarray, Tensor]*, ***kwargs: Any*) → Tuple[Tensor, Tensor]

Mapping: s1, act, s2 -> mse_loss, act_hat.

training: bool

Continuous

class tianshou.utils.net.continuous.**Actor**(*preprocess_net: Module*, *action_shape: Sequence[int]*, *hidden_sizes: Sequence[int] = ()*, *max_action: float = 1.0*, *device: Union[str, int, device] = 'cpu'*, *preprocess_net_output_dim: Optional[int] = None*)

Bases: Module

Simple actor network. Will create an actor operated in continuous action space with structure of preprocess_net —> action_shape.

Parameters

- **preprocess_net** – a self-defined preprocess_net which output a flattened hidden state.
- **action_shape** – a sequence of int for the shape of action.
- **hidden_sizes** – a sequence of int for constructing the MLP after preprocess_net. Default to empty sequence (where the MLP now contains only a single linear layer).
- **max_action** (*float*) – the scale for the final action logits. Default to 1.
- **preprocess_net_output_dim** (*int*) – the output dimension of preprocess_net.

For advanced usage (how to customize the network), please refer to *Build the Network*.

See also:

Please refer to *Net* as an instance of how preprocess_net is suggested to be defined.

forward(*obs: Union[ndarray, Tensor]*, *state: Optional[Any] = None*, *info: Dict[str, Any] = {}*) → Tuple[Tensor, Any]

Mapping: obs -> logits -> action.

training: bool

class tianshou.utils.net.continuous.**Critic**(*preprocess_net: ~torch.nn.modules.module.Module*, *hidden_sizes: ~typing.Sequence[int] = ()*, *device: ~typing.Union[str, int, ~torch.device] = 'cpu'*, *preprocess_net_output_dim: ~typing.Optional[int] = None*, *linear_layer: ~typing.Type[~torch.nn.modules.linear.Linear] = <class 'torch.nn.modules.linear.Linear'>*, *flatten_input: bool = True*)

Bases: Module

Simple critic network. Will create an actor operated in continuous action space with structure of preprocess_net —> 1(q value).

Parameters

- **preprocess_net** – a self-defined preprocess_net which output a flattened hidden state.
- **hidden_sizes** – a sequence of int for constructing the MLP after preprocess_net. Default to empty sequence (where the MLP now contains only a single linear layer).
- **preprocess_net_output_dim** (*int*) – the output dimension of preprocess_net.
- **linear_layer** – use this module as linear layer. Default to nn.Linear.
- **flatten_input** (*bool*) – whether to flatten input data for the last layer. Default to True.

For advanced usage (how to customize the network), please refer to *Build the Network*.

See also:

Please refer to *Net* as an instance of how preprocess_net is suggested to be defined.

forward(*obs: Union[ndarray, Tensor]*, *act: Optional[Union[ndarray, Tensor]] = None*, *info: Dict[str, Any] = {}*) → Tensor

Mapping: (s, a) -> logits -> Q(s, a).

training: bool

class tianshou.utils.net.continuous.**ActorProb**(*preprocess_net: Module*, *action_shape: Sequence[int]*, *hidden_sizes: Sequence[int] = ()*, *max_action: float = 1.0*, *device: Union[str, int, device] = 'cpu'*, *unbounded: bool = False*, *conditioned_sigma: bool = False*, *preprocess_net_output_dim: Optional[int] = None*)

Bases: Module

Simple actor network (output with a Gauss distribution).

Parameters

- **preprocess_net** – a self-defined preprocess_net which output a flattened hidden state.
- **action_shape** – a sequence of int for the shape of action.
- **hidden_sizes** – a sequence of int for constructing the MLP after preprocess_net. Default to empty sequence (where the MLP now contains only a single linear layer).
- **max_action** (*float*) – the scale for the final action logits. Default to 1.
- **unbounded** (*bool*) – whether to apply tanh activation on final logits. Default to False.
- **conditioned_sigma** (*bool*) – True when sigma is calculated from the input, False when sigma is an independent parameter. Default to False.

- **preprocess_net_output_dim** (*int*) – the output dimension of preprocess_net.

For advanced usage (how to customize the network), please refer to *Build the Network*.

See also:

Please refer to *Net* as an instance of how preprocess_net is suggested to be defined.

forward(*obs: Union[ndarray, Tensor]*, *state: Optional[Any] = None*, *info: Dict[str, Any] = {}*) → Tuple[Tuple[Tensor, Tensor], Any]

Mapping: obs -> logits -> (mu, sigma).

training: bool

class tianshou.utils.net.continuous.**RecurrentActorProb**(*layer_num: int*, *state_shape: Sequence[int]*, *action_shape: Sequence[int]*, *hidden_layer_size: int = 128*, *max_action: float = 1.0*, *device: Union[str, int, device] = 'cpu'*, *unbounded: bool = False*, *conditioned_sigma: bool = False*)

Bases: Module

Recurrent version of ActorProb.

For advanced usage (how to customize the network), please refer to *Build the Network*.

forward(*obs: Union[ndarray, Tensor]*, *state: Optional[Dict[str, Tensor]] = None*, *info: Dict[str, Any] = {}*) → Tuple[Tuple[Tensor, Tensor], Dict[str, Tensor]]

Almost the same as *Recurrent*.

training: bool

class tianshou.utils.net.continuous.**RecurrentCritic**(*layer_num: int*, *state_shape: Sequence[int]*, *action_shape: Sequence[int] = [0]*, *device: Union[str, int, device] = 'cpu'*, *hidden_layer_size: int = 128*)

Bases: Module

Recurrent version of Critic.

For advanced usage (how to customize the network), please refer to *Build the Network*.

forward(*obs: Union[ndarray, Tensor]*, *act: Optional[Union[ndarray, Tensor]] = None*, *info: Dict[str, Any] = {}*) → Tensor

Almost the same as *Recurrent*.

training: bool

class tianshou.utils.net.continuous.**Perturbation**(*preprocess_net: Module*, *max_action: float*, *device: Union[str, int, device] = 'cpu'*, *phi: float = 0.05*)

Bases: Module

Implementation of perturbation network in BCQ algorithm. Given a state and action, it can generate perturbed action.

Parameters

- **preprocess_net** (*torch.nn.Module*) – a self-defined preprocess_net which output a flattened hidden state.
- **max_action** (*float*) – the maximum value of each dimension of action.

- **device** (*Union[str, int, torch.device]*) – which device to create this model on. Default to cpu.
- **phi** (*float*) – max perturbation parameter for BCQ. Default to 0.05.

For advanced usage (how to customize the network), please refer to *Build the Network*.

See also:

You can refer to *examples/offline/offline_bcq.py* to see how to use it.

forward(*state: Tensor*, *action: Tensor*) → Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class tianshou.utils.net.continuous.**VAE**(*encoder: Module*, *decoder: Module*, *hidden_dim: int*, *latent_dim: int*, *max_action: float*, *device: Union[str, device] = 'cpu'*)

Bases: `Module`

Implementation of VAE. It models the distribution of action. Given a state, it can generate actions similar to those in batch. It is used in BCQ algorithm.

Parameters

- **encoder** (*torch.nn.Module*) – the encoder in VAE. Its input_dim must be state_dim + action_dim, and output_dim must be hidden_dim.
- **decoder** (*torch.nn.Module*) – the decoder in VAE. Its input_dim must be state_dim + latent_dim, and output_dim must be action_dim.
- **hidden_dim** (*int*) – the size of the last linear-layer in encoder.
- **latent_dim** (*int*) – the size of latent layer.
- **max_action** (*float*) – the maximum value of each dimension of action.
- **device** (*Union[str, torch.device]*) – which device to create this model on. Default to "cpu".

For advanced usage (how to customize the network), please refer to *Build the Network*.

See also:

You can refer to *examples/offline/offline_bcq.py* to see how to use it.

forward(*state: Tensor*, *action: Tensor*) → Tuple[Tensor, Tensor, Tensor]

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

decode(*state: Tensor*, *latent_z: Optional[Tensor] = None*) → Tensor

training: bool

1.15 Contributing to Tianshou

1.15.1 Install Develop Version

To install Tianshou in an "editable" mode, run

```
$ pip install -e ".[dev]"
```

in the main directory. This installation is removable by

```
$ python setup.py develop --uninstall
```

1.15.2 PEP8 Code Style Check and Code Formatter

Please set up pre-commit by running

```
$ pre-commit install
```

in the main directory. This should make sure that your contribution is properly formatted before every commit.

We follow PEP8 python code style with flake8. To check, in the main directory, run:

```
$ make lint
```

We use isort and yapf to format all codes. To format, in the main directory, run:

```
$ make format
```

To check if formatted correctly, in the main directory, run:

```
$ make check-codestyle
```

1.15.3 Type Check

We use mypy to check the type annotations. To check, in the main directory, run:

```
$ make mypy
```

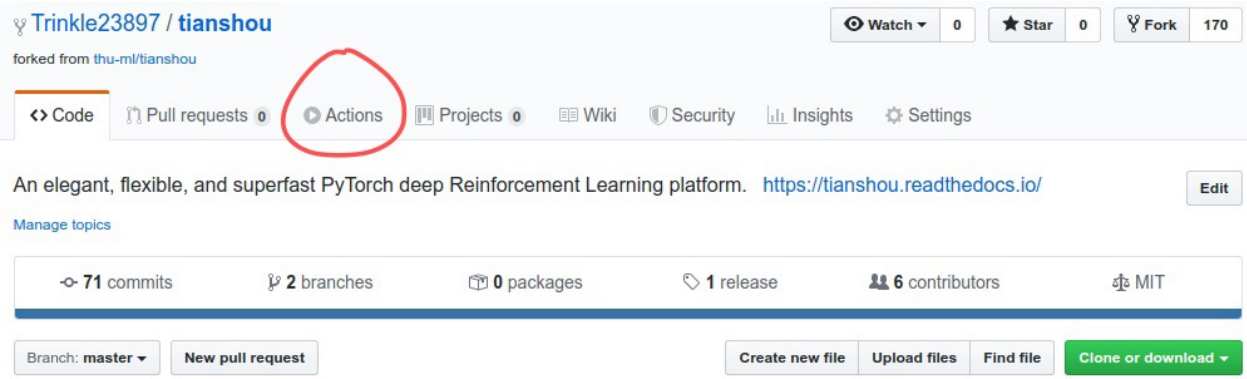
1.15.4 Test Locally

This command will run automatic tests in the main directory

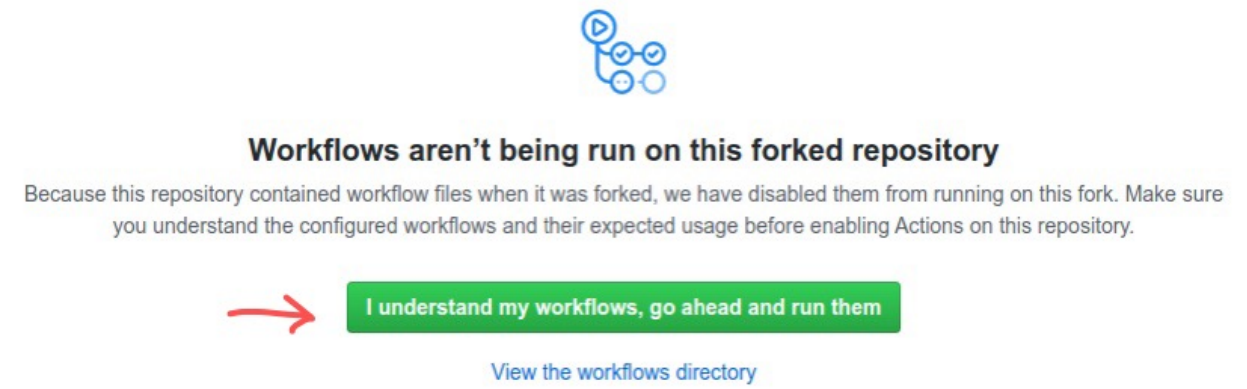
```
$ make pytest
```

1.15.5 Test by GitHub Actions

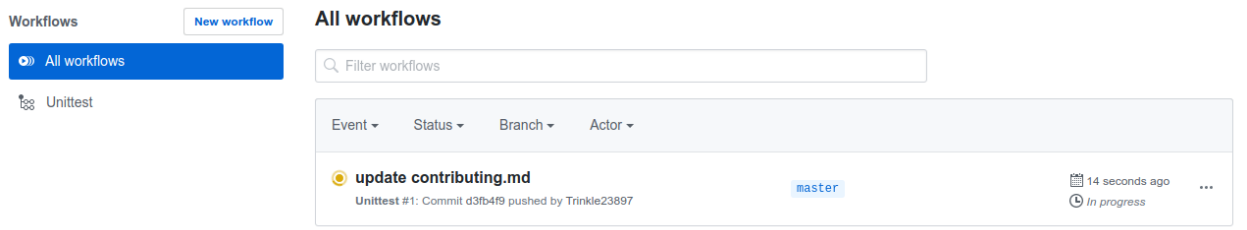
1. Click the `Actions` button in your own repo:



2. Click the green button:



3. You will see `Actions Enabled.` on the top of html page.
4. When you push a new commit to your own repo (e.g. `git push`), it will automatically run the test in this page:



1.15.6 Documentation

Documentations are written under the `docs/` directory as ReStructuredText (`.rst`) files. `index.rst` is the main page. A Tutorial on ReStructuredText can be found here.

API References are automatically generated by Sphinx according to the outlines under `docs/api/` and should be modified when any code changes.

To compile documentation into webpage, run

```
$ make doc
```

The generated webpage is in `docs/_build` and can be viewed with browser (http://0.0.0.0:8000/).

Chinese documentation is in https://tianshou.readthedocs.io/zh/latest/.

1.15.7 Documentation Generation Test

We have the following three documentation tests:

1. pydocstyle: test all docstring under `tianshou/`;
2. doc8: test ReStructuredText format;
3. sphinx test: test if there is any error/warning when generating front-end html documentation.

To check, in the main directory, run:

```
$ make check-docstyle
```

1.16 Contributor

We always welcome contributions to help make Tianshou better. Below are an incomplete list of our contributors (find more on this page).

- Jiayi Weng (Trinkle23897)
- Alexis Duburcq (duburcqa)
- Kaichao You (youkaichao)
- Huayu Chen (ChenDRAG)
- Yi Su (nuance1979)

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. URL: https://doi.org/10.1038/nature14236, doi:10.1038/nature14236.

[LHP+16] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. 2016. URL: http://arxiv.org/abs/1509.02971.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, 2017. URL: http://arxiv.org/abs/1707.06347, arXiv:1707.06347.

PYTHON MODULE INDEX

t

INDEX

Symbols

A

B

C

D

E

F

G

H

I

Q

R

S

T

U

V

W