Tianshou

Release 0.3.0

Tianshou contributors

Oct 08, 2020

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:

- `PGPolicy` Policy Gradient
- `DQNPolicy` Deep Q-Network
- `DQNPolicy` Double DQN
- `DQNPolicy` Dueling DQN
- `A2CPolicy` Advantage Actor-Critic
- `DDPGPolicy` Deep Deterministic Policy Gradient
- `PPOPolicy` Proximal Policy Optimization
- `TD3Policy` Twin Delayed DDPG
- `SACPolicy` Soft Actor-Critic
- `DiscreteSACPolicy` Discrete Soft Actor-Critic
- `PSRLPolicy` Posterior Sampling Reinforcement Learning
- `ImitationPolicy` Imitation Learning
- `PrioritizedReplayBuffer` Prioritized Experience Replay
- `compute_episodic_return()` Generalized Advantage Estimator

Here is Tianshou's other features:

- Elegant framework, using only ~2000 lines of code
- Support parallel environment simulation (synchronous or asynchronous) for all algorithms: *Parallel Sampling*
- 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*
- Comprehensive unit tests, including functional checking, RL pipeline checking, documentation checking, PEP8 code-style checking, and type checking

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

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 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.

1.1.1 Make an Environment

First of all, you have to make an environment for your agent to interact with. For environment interfaces, we follow the convention of OpenAI Gym. In your Python code, simply import Tianshou and make the environment:

```
import gym
import tianshou as ts

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

CartPole-v0 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, depending on the probability distribution on the action.

1.1.2 Setup Multi-environment Wrapper

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

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

Tianshou supports parallel sampling for all algorithms. It provides four types of vectorized environment wrapper: *DummyVectorEnv*, *SubprocVectorEnv*, *ShmemVectorEnv*, and *RayVectorEnv*. 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(8)])
test_envs = ts.env.DummyVectorEnv([lambda: gym.make('CartPole-v0') for _ in␣
↪range(100)])
```

Here, we set up 8 environments in `train_envs` and 100 environments in `test_envs`.

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.

1.1.3 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(
```

(continues on next page)

(continued from previous page)

```
            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)),
        )

    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)
```

It is also possible to 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.

1.1.4 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)
```

1.1.5 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.

```
train_collector = ts.data.Collector(policy, train_envs, ts.data.
↪ReplayBuffer(size=20000))
test_collector = ts.data.Collector(policy, test_envs)
```

1.1.6 Train Policy with a Trainer

Tianshou provides *onpolicy_trainer* and *offpolicy_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=1000, collect_per_step=10,
    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,
    writer=None)
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 step for updating policy network in one epoch;
- `collect_per_step`: The number of frames the collector would collect before the network update. For example, the code above means "collect 10 frames 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.
- `writer`: See below.

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

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

Pass the writer 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',
```

(continues on next page)

(continued from previous page)

```
    '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.

1.1.7 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'))
```

1.1.8 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)
collector.collect(n_episode=1, render=1 / 35)
```

1.1.9 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:

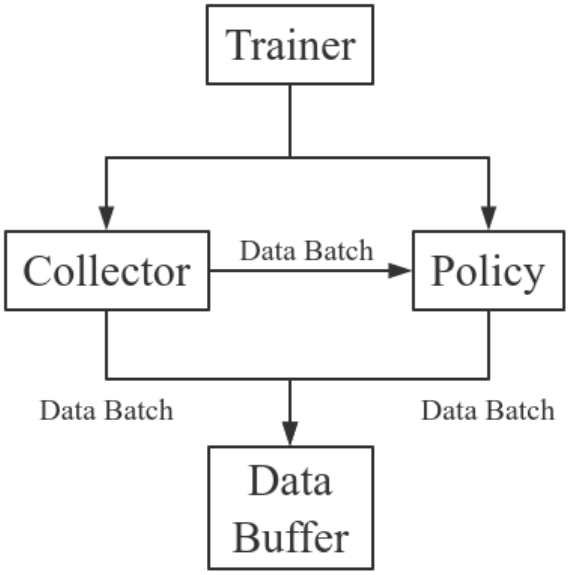
```
# pre-collect at least 5000 frames with random action before training
policy.set_eps(1)
train_collector.collect(n_step=5000)

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['rew'] >= env.spec.reward_threshold or i % 1000 == 0:
        policy.set_eps(0.05)
        result = test_collector.collect(n_episode=100)
        if result['rew'] >= env.spec.reward_threshold:
            print(f'Finished training! Test mean returns: {result["rew"]}')
            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*.

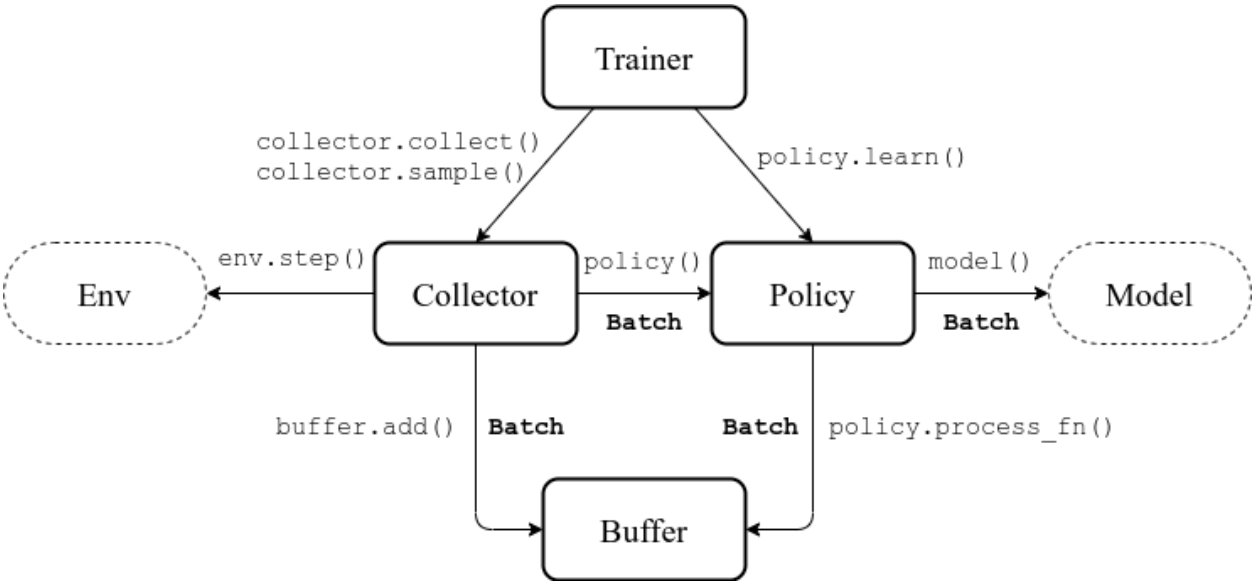
References

1.2 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:



1.2.1 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])
>>> 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*.

1.2.2 Buffer

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

The current implementation of Tianshou typically use 7 reserved keys in *Batch*:

- `obs` the observation of step t ;
- `act` the action of step t ;
- `rew` the reward of step t ;
- `done` the done flag of step t ;
- `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;

The following code snippet illustrates its usage:

```
>>> import pickle, numpy as np
>>> from tianshou.data import ReplayBuffer
>>> buf = ReplayBuffer(size=20)
```

(continues on next page)

(continued from previous page)

```
>>> for i in range(3):
...     buf.add(obs=i, act=i, rew=i, done=i, 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
>>> pickle.dump(buf, open('buf.pkl', 'wb'))  # save to file "buf.pkl"
>>> buf2 = ReplayBuffer(size=10)
>>> for i in range(15):
...     buf2.add(obs=i, act=i, rew=i, done=i, 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)
array([ 0.,  1.,  2.,  5.,  6.,  7.,  8.,  9., 10., 11., 12., 13., 14.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.])

>>> # get a random sample from buffer
>>> # the batch_data is equal to buf[incide].
>>> batch_data, indice = buf.sample(batch_size=4)
>>> batch_data.obs == buf[indice].obs
array([ True,  True,  True,  True])
>>> len(buf)
13
>>> buf = pickle.load(open('buf.pkl', 'rb'))  # load from "buf.pkl"
>>> 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):
...     done = i % 5 == 0
...     buf.add(obs={'id': i}, act=i, rew=i, done=done,
...             obs_next={'id': i + 1})
>>> print(buf)  # you can see obs_next is not saved in buf
ReplayBuffer(
    act: array([ 9., 10., 11., 12., 13., 14., 15.,  7.,  8.]),
    done: array([0., 1., 0., 0., 0., 0., 1., 0., 0.]),
    info: Batch(),
    obs: Batch(
             id: array([ 9., 10., 11., 12., 13., 14., 15.,  7.,  8.]),
         ),
    policy: Batch(),
    rew: array([ 9., 10., 11., 12., 13., 14., 15.,  7.,  8.]),
)
>>> index = np.arange(len(buf))
>>> print(buf.get(index, 'obs').id)
```

(continues on next page)

(continued from previous page)

```
[[ 7.  7.  8.  9.]
 [ 7.  8.  9. 10.]
 [11. 11. 11. 11.]
 [11. 11. 11. 12.]
 [11. 11. 12. 13.]
 [11. 12. 13. 14.]
 [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()
0.0
>>> # we can get obs_next through __getitem__, even if it doesn't exist
>>> print(buf[:].obs_next.id)
[[ 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.]
 [ 7.  7.  7.  8.]
 [ 7.  7.  8.  9.]]
```

param int size the size of replay buffer.

param int stack_num the frame-stack sampling argument, should be greater than or equal to 1, defaults to 1 (no stacking).

param bool ignore_obs_next whether to store obs_next, defaults to False.

param bool save_only_last_obs only save the last obs/obs_next when it has a shape of (timestep, ...) because of temporal stacking, defaults to False.

param bool sample_avail the parameter indicating sampling only available index when using frame-stack sampling method, defaults to False. This feature is not supported in Prioritized Replay Buffer currently.

Tianshou provides other type of data buffer such as *ListReplayBuffer* (based on list), *PrioritizedReplayBuffer* (based on Segment Tree and `numpy.ndarray`). Check out *ReplayBuffer* for more detail.

1.2.3 Policy

Tianshou aims to modularizing 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 frame'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
s = env.reset()
buffer = Buffer(size=10000)
agent = DQN()
for i in range(int(1e6)):
    a = agent.compute_action(s)
    s_, r, d, _ = env.step(a)
    buffer.store(s, a, s_, r, d)
    s = s_
    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, indice):
        buffer_len = len(buffer)
        batch_2 = buffer[(indice + 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[(indice + 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[(indice + 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*.

1.2.4 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 (at least) a specified number of step `n_step` or episode `n_episode` and store the data in the replay buffer.

Why do we mention **at least** here? For multiple environments, we could not directly store the collected data into the replay buffer, since it breaks the principle of storing data chronologically.

The proposed solution is to add some cache buffers inside the collector. Once collecting **a full episode of trajectory**, it will move the stored data from the cache buffer to the main buffer. To satisfy this condition, the collector will interact with environments that may exceed the given step number or episode number.

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

1.2.5 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 two types of trainer: *onpolicy_trainer()* and *offpolicy_trainer()*, corresponding to on-policy algorithms (such as Policy Gradient) and off-policy algorithms (such as DQN). Please check out *tianshou.trainer* for the usage.

1.2.6 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
s = 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
    a = agent.compute_action(s)                             # act = policy(batch,␣
↪...).act
    s_, r, d, _ = env.step(a)                               # collector.collect(..
↪.)
    buffer.store(s, a, s_, r, d)                            # collector.collect(..
↪.)
    s = s_                                                  # 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, indice =␣
↪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, indice)
        # update DQN policy
        agent.update(b_s, b_a, b_s_, b_r, b_d, b_ret)       # policy.learn(batch,␣
↪...)
```

1.2.7 Conclusion

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

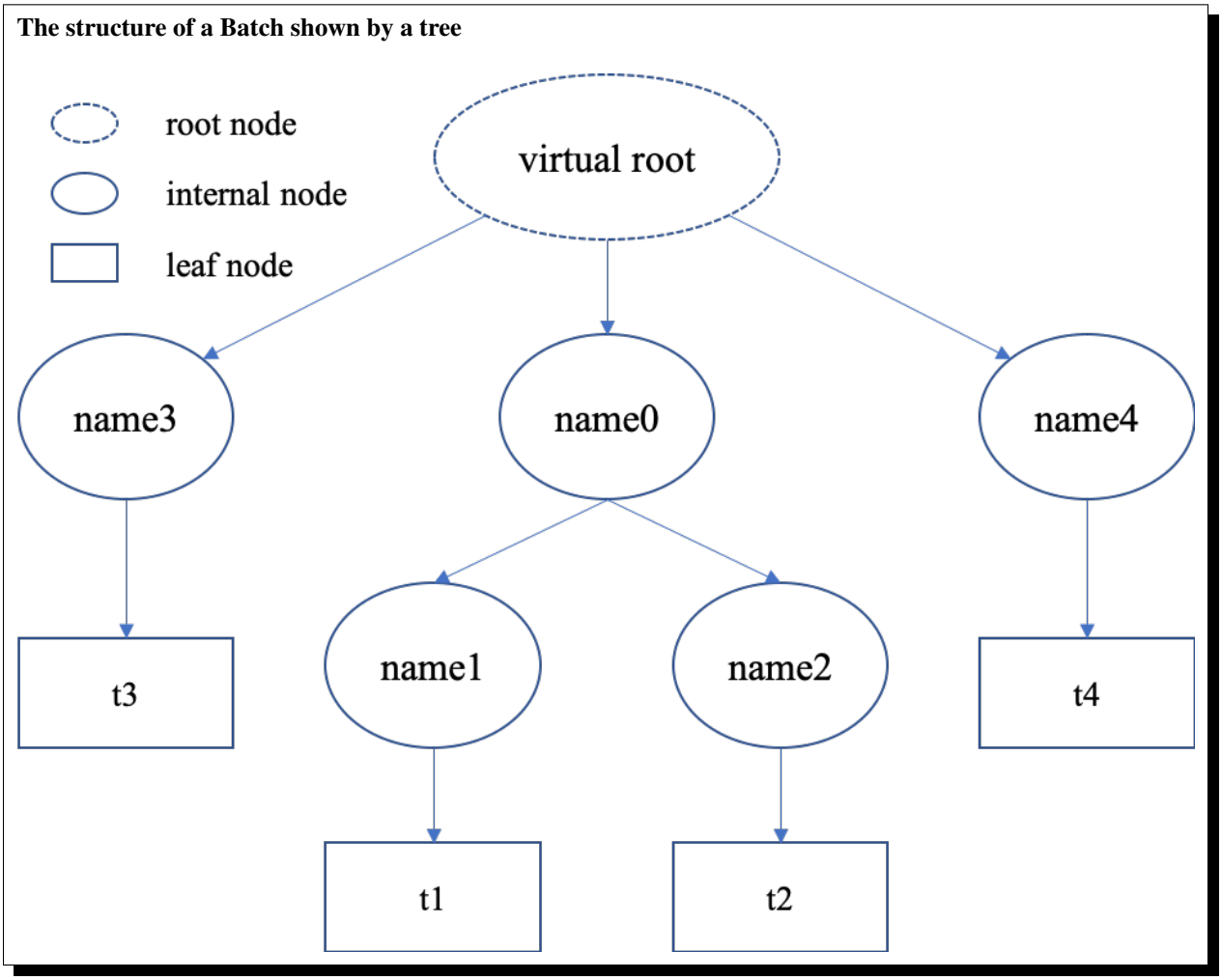
1.3 Understand Batch

`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`.

1.3.1 Hierarchical Named Tensors

The structure of a Batch shown by a tree



"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.

1.3.2 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 recurse 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 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)
[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]),
)
```

(continues on next page)

(continued from previous page)

```
>>> # 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]),
    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]),
```

(continues on next page)

(continued from previous page)

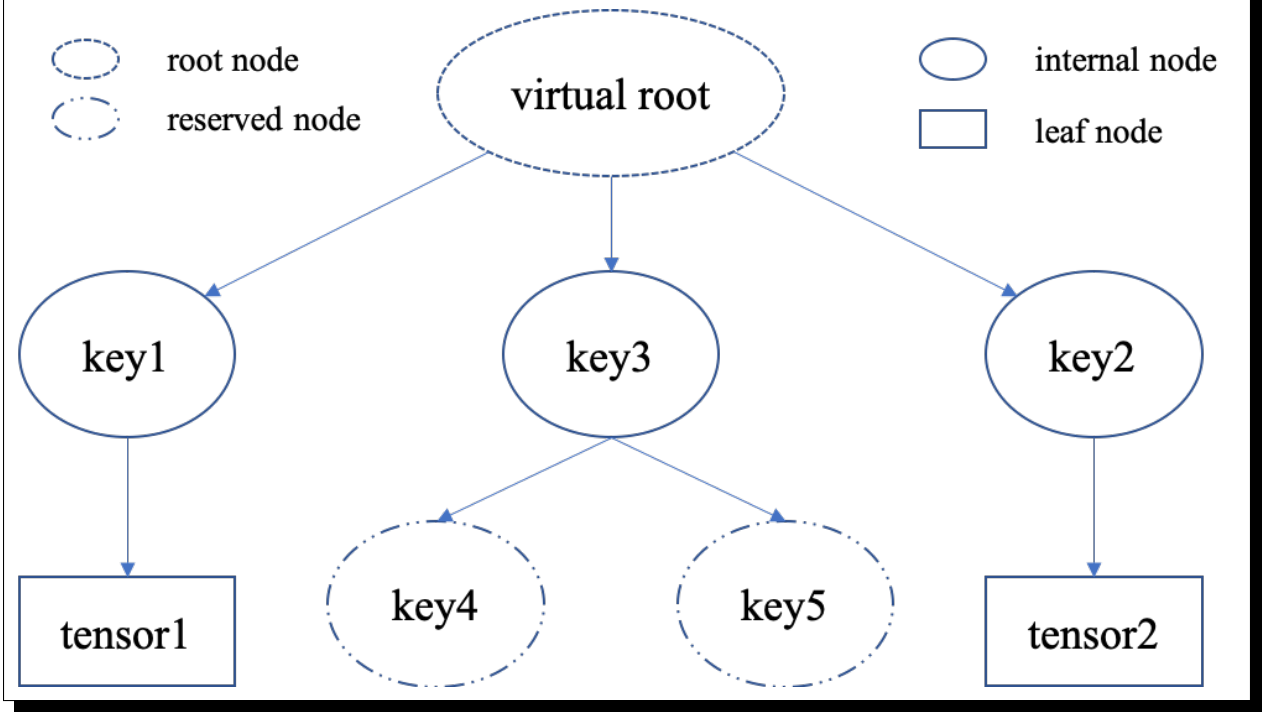
```
    a: array([[1., 3.]]),
)]
>>> data_cat = Batch.cat(data_split)
>>> print(data_cat)
Batch(
    b: array([ 5, -5]),
    a: array([[0., 2.],
              [1., 3.]]),
)
```

1.3.3 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
```

(continues on next page)

(continued from previous page)

```
# 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 recurse 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)`.

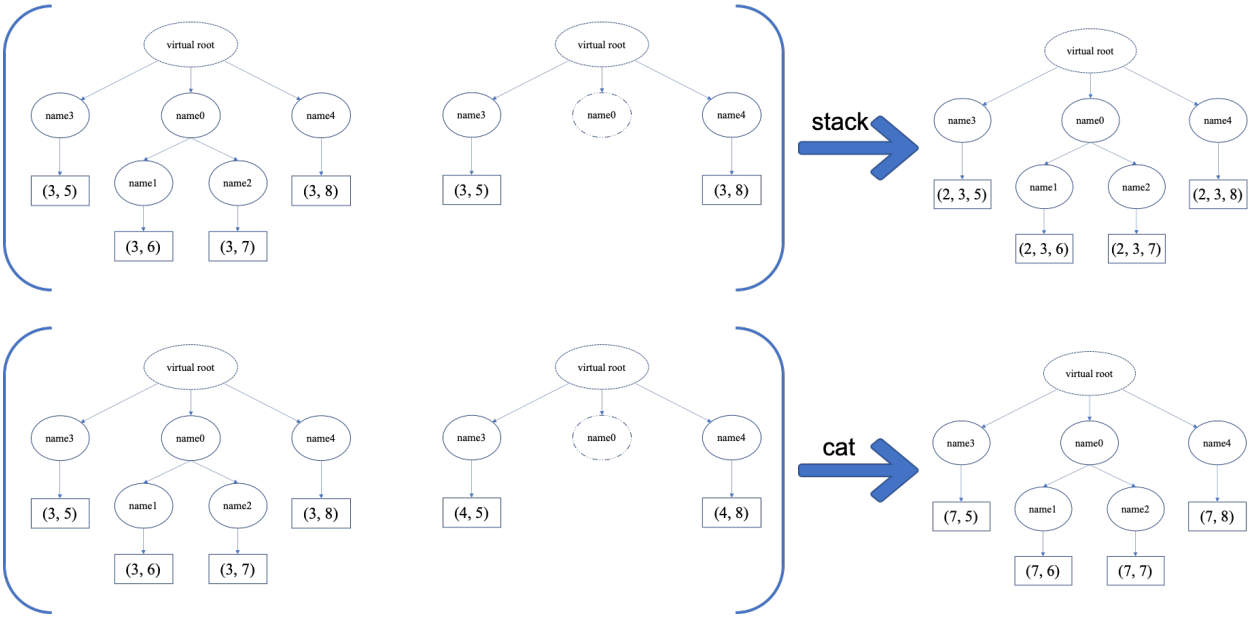
Besides, values of reserved keys are undetermined, so they have no length, neither. Or, to be specific, values of reserved keys have lengths of **any**. When there is a mix of tensors and reserved keys, the latter will be ignored in `len(obj)` and the minimum length of tensors is returned. When there 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])))
```

(continues on next page)

(continued from previous page)

```
>>> 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),
)
>>> # 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

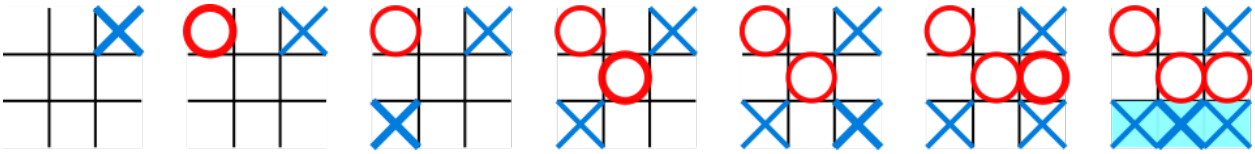
1. `Batch` is serializable and therefore Pickle compatible. `Batch` objects can be saved to disk and later restored by the python `pickle` module. This pickle compatibility is especially important for distributed sampling from environments.

```
>>> 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 NumPy ndarrays 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.

1.4 Multi-Agent RL

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.



1.4.1 Tic-Tac-Toe Environment

The scripts are located at `test/multiagent/`. We have implemented a Tic-Tac-Toe environment inherit the *`MultiAgentEnv`* that supports Tic-Tac-Toe of any scale. Let's first explore the environment. The 3x3 Tic-Tac-Toe is too easy, so we will focus on 6x6 Tic-Tac-Toe where 4 same signs in a row are considered to win.

```
>>> from tic_tac_toe_env import TicTacToeEnv     # the module tic_tac_toe_env is in␣
↪test/multiagent/
>>> board_size = 6                               # the size of board size
>>> win_size = 4                                 # how many signs in a row are␣
↪considered to win
>>>
>>> # This board has 6 rows and 6 cols (36 places in total)
```

(continues on next page)

(continued from previous page)

```
>>> # Players place 'x' and 'o' in turn on the board
>>> # The player who first gets 4 consecutive 'x's or 'o's wins
>>>
>>> env = TicTacToeEnv(size=board_size, win_size=win_size)
>>> obs = env.reset()
>>> env.render()                                    # render the empty board
board (step 0):
=================
===_ _ _ _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
=================
>>> print(obs)                                      # let's see the shape of the␣
↪observation
{'agent_id': 1,
 'obs': array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0]], dtype=int32),
 'mask': array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
                 True,  True,  True,  True,  True,  True,  True,  True,  True,
                 True,  True,  True,  True,  True,  True,  True,  True,  True,
                 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, where agent_id $\in [1, N]$, N is the number of agents. In our Tic-Tac-Toe case, N is 2. The agent_id starts at 1 because we reserve 0 for the environment itself. Sometimes the developer may want to control the behavior of the environment, for example, to determine how to dispatch cards in Poker.
- `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 (6, 6). The values can be "0/1/-1": 0 for empty, 1 for `x`, -1 for `o`. Agent 1 places `x` on the board, while agent 2 places `o` on the board.
- `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 `gym.spaces.Discrete` or `gym.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
```

(continues on next page)

(continued from previous page)

```
>>> obs, reward, done, info = env.step(action)  # the env.step follows the api of
↪OpenAI Gym
>>> print(obs)                                  # notice the change in the observation
{'agent_id': 2,
 'obs': array([[1, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0]], dtype=int32),
 'mask': array([False,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  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)
{}
```

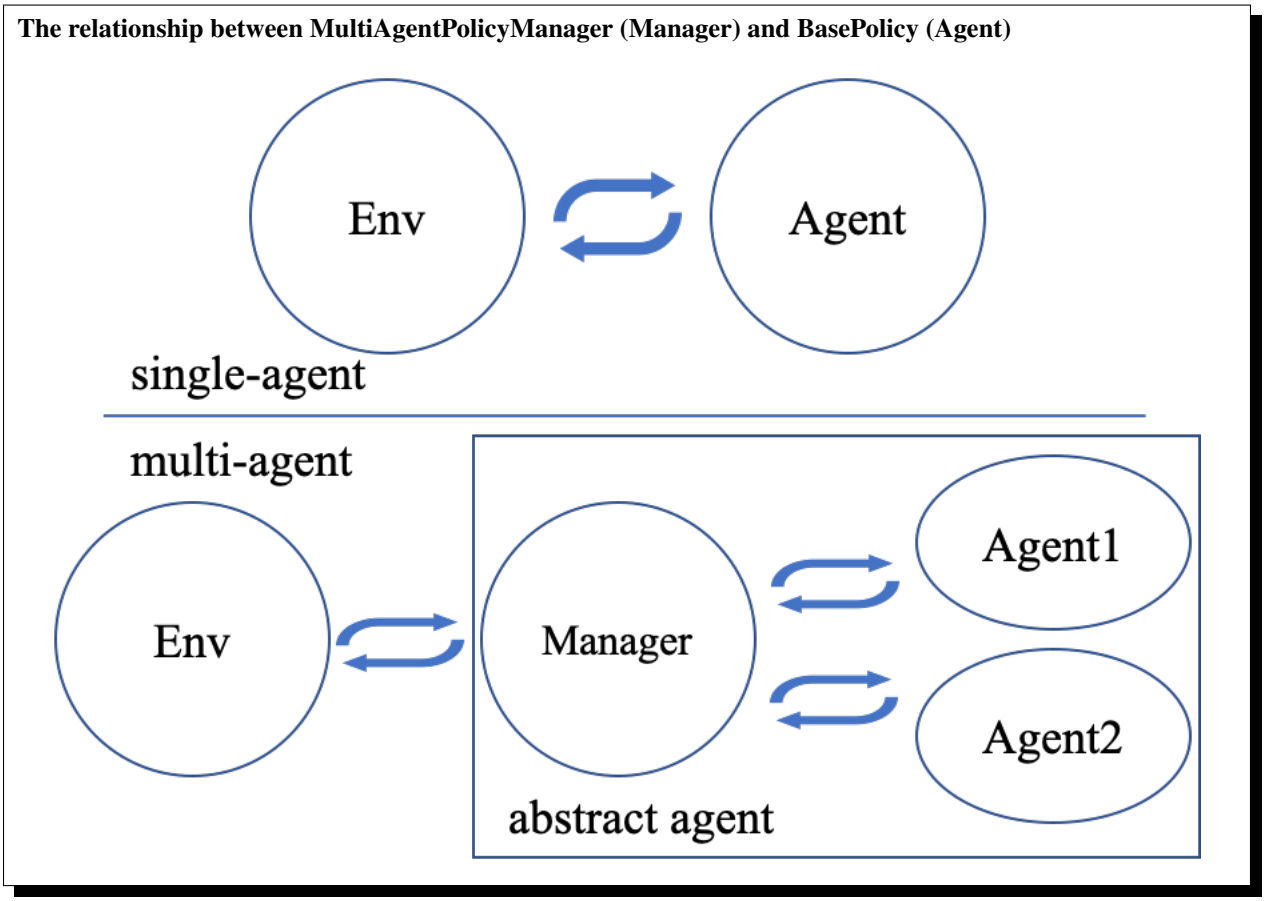
One worth-noting case is that the game is over when there is only one empty position, rather than when there is no position. This is because the player just has one choice (literally no choice) in this game.

```
>>> # omitted actions: 6, 1, 7, 2, 8
>>> obs, reward, done, info = env.step(3)  # player 1 wins
>>> print((reward, done))
(array([ 1., -1.], dtype=float32), array(True))
>>> env.render()                                # 'X' and 'O' indicate the last action
board (step 7):
=================
===x x x X _ _===
===o o o _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
=================
```

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

1.4.2 Two Random Agent

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.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()])
>>>
>>> # 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, reward_metric=lambda x: x[0])
>>>
>>> # 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)
board (step 20):
=================
===o x _ o o o===
===_ _ x _ _ x===
```

(continues on next page)

(continued from previous page)

```
===x _ o o x _===
===O _ o o x _===
===x _ o _ _ _===
===x _ _ _ x x===
=================
board (step 21):
=================
===o x _ o o o===
===_ _ x _ _ x===
===x _ o o x _===
===o _ o o x _===
===x _ o X _ _===
===x _ _ _ x x===
=================
board (step 22):
=================
===o x _ o o o===
===_ O x _ _ x===
===x _ o o x _===
===o _ o o x _===
===x _ o x _ _===
===x _ _ _ x x===
=================
```

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.

1.4.3 Train an MARL Agent

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

```
import os
import torch
import argparse
import numpy as np
from copy import deepcopy
from torch.utils.tensorboard import SummaryWriter

from tianshou.env import DummyVectorEnv
from tianshou.utils.net.common import Net
from tianshou.trainer import offpolicy_trainer
from tianshou.data import Collector, ReplayBuffer
from tianshou.policy import BasePolicy, RandomPolicy, DQNPolicy,␣
↪MultiAgentPolicyManager

from tic_tac_toe_env import TicTacToeEnv
```

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.

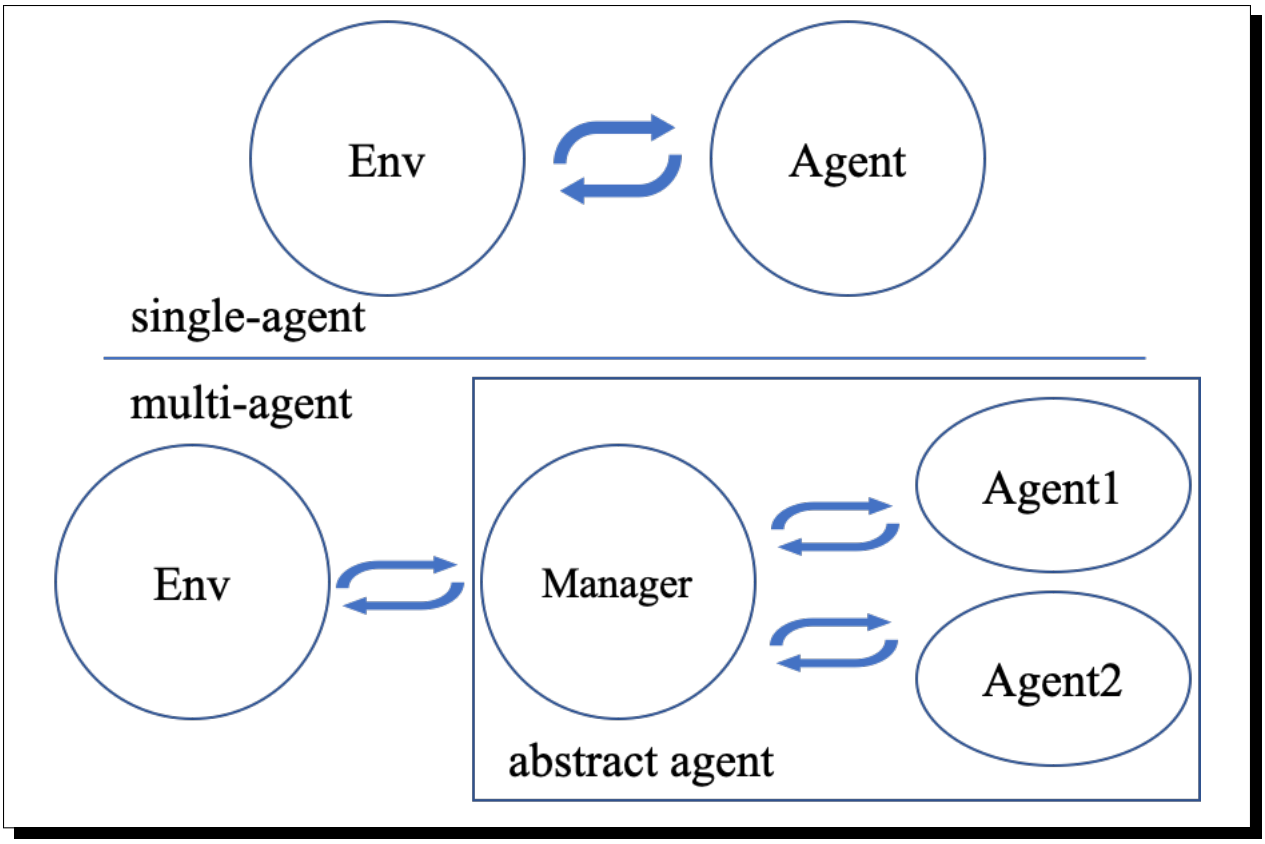
```
def get_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('--seed', type=int, default=1626)
    parser.add_argument('--eps-test', type=float, default=0.05)
    parser.add_argument('--eps-train', type=float, default=0.1)
```

(continues on next page)

(continued from previous page)

```
    parser.add_argument('--buffer-size', type=int, default=20000)
    parser.add_argument('--lr', type=float, default=1e-3)
    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=10)
    parser.add_argument('--step-per-epoch', type=int, default=1000)
    parser.add_argument('--collect-per-step', type=int, default=10)
    parser.add_argument('--batch-size', type=int, default=64)
    parser.add_argument('--layer-num', type=int, default=3)
    parser.add_argument('--training-num', type=int, default=8)
    parser.add_argument('--test-num', type=int, default=100)
    parser.add_argument('--logdir', type=str, default='log')
    parser.add_argument('--render', type=float, default=0.1)
    parser.add_argument('--board_size', type=int, default=6)
    parser.add_argument('--win_size', type=int, default=4)
    parser.add_argument('--win-rate', type=float, default=np.float32(0.9),
                        help='the expected winning rate')
    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.')
    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.parse_args()
```

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=get_args(),
               agent_learn=None,      # BasePolicy
               agent_opponent=None,   # BasePolicy
               optim=None,            # torch.optim.Optimizer
               ):  # return a tuple of (BasePolicy, torch.optim.Optimizer)
    env = TicTacToeEnv(args.board_size, args.win_size)
    args.state_shape = env.observation_space.shape or env.observation_space.n
    args.action_shape = env.action_space.shape or env.action_space.n

    if agent_learn is None:
        net = Net(args.layer_num, args.state_shape, args.action_shape, args.device).
↪to(args.device)
```

(continues on next page)

(continued from previous page)

```
        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()

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

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.

```
args = get_args()
# the reward is a vector, we need a scalar metric to monitor the training.
# we choose the reward of the learning agent
Collector._default_rew_metric = lambda x: x[args.agent_id - 1]

# ======== a test function that tests a pre-trained agent and exit ======
def watch(args=get_args(),
          agent_learn=None,      # BasePolicy
          agent_opponent=None):  # BasePolicy
    env = TicTacToeEnv(args.board_size, args.win_size)
    policy, optim = get_agents(
        args, agent_learn=agent_learn, agent_opponent=agent_opponent)
    policy.eval()
    policy.policies[args.agent_id - 1].set_eps(args.eps_test)
    collector = Collector(policy, env)
    result = collector.collect(n_episode=1, render=args.render)
    print(f'Final reward: {result["rew"]}, length: {result["len"]}')
if args.watch:
    watch(args)
    exit(0)

# ======== environment setup =========
env_func = lambda: TicTacToeEnv(args.board_size, args.win_size)
train_envs = DummyVectorEnv([env_func for _ in range(args.training_num)])
test_envs = DummyVectorEnv([env_func 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)
```

(continues on next page)

(continued from previous page)

```
# ======== agent setup =========
policy, optim = get_agents()

# ======== collector setup =========
train_collector = Collector(policy, train_envs, ReplayBuffer(args.buffer_size))
test_collector = Collector(policy, test_envs)
train_collector.collect(n_step=args.batch_size)

# ======== tensorboard logging setup =========
if not hasattr(args, 'writer'):
    log_path = os.path.join(args.logdir, 'tic_tac_toe', 'dqn')
    writer = SummaryWriter(log_path)
else:
    writer = args.writer

# ======== callback functions used during training =========

def save_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[args.agent_id - 1].state_dict(),
        model_save_path)

def stop_fn(mean_rewards):
    return mean_rewards >= args.win_rate  # 95% winning rate by default
    # the default args.win_rate is 0.9, but the reward is [-1, 1]
    # instead of [0, 1], so args.win_rate == 0.9 is equal to 95% win rate.

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

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

# start training, this may require about three minutes
result = offpolicy_trainer(
    policy, train_collector, test_collector, args.epoch,
    args.step_per_epoch, args.collect_per_step, args.test_num,
    args.batch_size, train_fn=train_fn, test_fn=test_fn,
    stop_fn=stop_fn, save_fn=save_fn, writer=writer,
    test_in_train=False)

agent = policy.policies[args.agent_id - 1]
# let's watch the match!
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:

```
board (step 1):
=================
===_ _ _ X _ _===
```

(continues on next page)

(continued from previous page)

```
===_ _ _ _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
=================
board (step 2):
=================
===_ _ _ x _ _===
===_ _ _ _ _ _===
===_ _ O _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
=================
board (step 3):
=================
===_ _ _ x _ _===
===_ _ _ _ _ _===
===_ _ o _ _ _===
===_ _ _ _ _ _===
===_ _ _ X _ _===
===_ _ _ _ _ _===
=================
board (step 4):
=================
===_ _ _ x _ _===
===_ _ _ _ _ _===
===_ _ o _ _ _===
===_ _ _ _ _ _===
===_ _ _ x _ _===
===_ _ O _ _ _===
=================
board (step 5):
=================
===_ _ _ x _ _===
===_ _ _ _ X _===
===_ _ o _ _ _===
===_ _ _ _ _ _===
===_ _ _ x _ _===
===_ _ o _ _ _===
=================
board (step 6):
=================
===_ _ _ x _ _===
===_ _ _ _ x _===
===_ _ o _ _ _===
===_ _ _ _ _ _===
===_ _ O x _ _===
===_ _ o _ _ _===
=================
board (step 7):
=================
===_ _ _ x _ X===
===_ _ _ _ x _===
===_ _ o _ _ _===
===_ _ _ _ _ _===
```

(continues on next page)

(continued from previous page)

```
===_ _ o x _ _===
===_ _ o _ _ _===
=================
board (step 8):
=================
===_ _ _ x _ x===
===_ _ _ _ x _===
===_ _ o _ _ _===
===_ _ _ _ O _===
===_ _ o x _ _===
===_ _ o _ _ _===
=================
board (step 9):
=================
===_ _ _ x _ x===
===_ _ _ _ x _===
===_ _ o _ _ _===
===_ _ _ _ o _===
===X _ o x _ _===
===_ _ o _ _ _===
=================
board (step 10):
=================
===_ _ _ x _ x===
===_ _ _ _ x _===
===_ _ o _ _ _===
===_ _ O _ o _===
===x _ o x _ _===
===_ _ o _ _ _===
=================
Final reward: 1.0, length: 10.0
```

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/multiagent/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:

```
board (step 1):
=================
===_ _ _ _ _ _===
===_ _ _ _ _ _===
===_ _ X _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
```

(continues on next page)

(continued from previous page)

```
=================
board (step 2):
=================
===_ _ _ _ _ _===
===_ _ _ _ _ _===
===_ _ x _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
===_ _ O _ _ _===
=================
board (step 3):
=================
===_ _ _ _ _ _===
===_ _ X _ _ _===
===_ _ x _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
===_ _ o _ _ _===
=================
board (step 4):
=================
===_ _ _ _ _ _===
===_ _ x _ _ _===
===_ _ x _ _ _===
===_ _ _ _ _ _===
===_ _ _ _ _ _===
===_ _ o O _ _===
=================
board (step 5):
=================
===_ _ _ _ _ _===
===_ _ x _ _ _===
===_ _ x _ _ _===
===_ _ _ _ _ _===
===_ _ _ X _ _===
===_ _ o o _ _===
=================
board (step 6):
=================
===_ _ _ _ _ _===
===_ _ x _ _ _===
===_ _ x _ _ _===
===_ _ _ _ _ _===
===_ _ _ x _ _===
===_ _ o o O _===
=================
board (step 7):
=================
===_ _ _ _ _ _===
===_ _ x _ X _===
===_ _ x _ _ _===
===_ _ _ _ _ _===
===_ _ _ x _ _===
===_ _ o o o _===
=================
board (step 8):
=================
```

(continues on next page)

(continued from previous page)

```
===_ _ _ _ _ _===
===_ _ x _ x _===
===_ _ x _ _ _===
===O _ _ _ _ _===
===_ _ _ x _ _===
===_ _ o o o _===
==================
board (step 9):
==================
===_ _ _ _ _ _===
===_ _ x _ x _===
===_ _ x _ _ _===
===o _ _ X _ _===
===_ _ _ x _ _===
===_ _ o o o _===
==================
board (step 10):
==================
===_ O _ _ _ _===
===_ _ x _ x _===
===_ _ x _ _ _===
===o _ _ x _ _===
===_ _ _ x _ _===
===_ _ o o o _===
==================
board (step 11):
==================
===_ o _ _ _ _===
===_ _ x _ x _===
===_ _ x _ _ X===
===o _ _ x _ _===
===_ _ _ x _ _===
===_ _ o o o _===
==================
board (step 12):
==================
===_ o O _ _ _===
===_ _ x _ x _===
===_ _ x _ _ x===
===o _ _ x _ _===
===_ _ _ x _ _===
===_ _ o o o _===
==================
board (step 13):
==================
===_ o o _ _ _===
===_ _ x _ x _===
===_ _ x _ _ x===
===o _ _ x X _===
===_ _ _ x _ _===
===_ _ o o o _===
==================
board (step 14):
==================
===O o o _ _ _===
===_ _ x _ x _===
===_ _ x _ _ x===
```

(continues on next page)

(continued from previous page)

```
===o _ _ x x _===
===_ _ _ x _ _===
===_ _ o o o _===
================
board (step 15):
================
===o o o _ _ _===
===_ _ x _ x _===
===_ _ x _ _ x===
===o _ _ x x _===
===X _ _ x _ _===
===_ _ o o o _===
================
board (step 16):
================
===o o o _ _ _===
===_ O x _ x _===
===_ _ x _ _ x===
===o _ _ x x _===
===x _ _ x _ _===
===_ _ o o o _===
================
board (step 17):
================
===o o o _ _ _===
===_ o x _ x _===
===_ _ x _ _ x===
===o _ _ x x _===
===x _ X x _ _===
===_ _ o o o _===
================
board (step 18):
================
===o o o _ _ _===
===_ o x _ x _===
===_ _ x _ _ x===
===o _ _ x x _===
===x _ x x _ _===
===_ O o o o _===
================
```

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!

1.5 Train a model-free RL agent within 30s

This page summarizes some hyper-parameter tuning experience and code-level trick when training a model-free DRL agent.

You can also contribute to this page with your own tricks :)

1.5.1 Avoid batch-size = 1

In the traditional RL training loop, we always use the policy to interact with only one environment for collecting data. That means most of the time the network use batch-size = 1. Quite inefficient! Here is an example of showing how inefficient it is:

```
import torch, time
from torch import nn

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(3, 128), nn.ReLU(inplace=True),
            nn.Linear(128, 128), nn.ReLU(inplace=True),
            nn.Linear(128, 1))
    def forward(self, s):
        return self.model(s)

net = Net()
cnt = 1000
div = 128
a = torch.randn([128, 3])

t = time.time()
for i in range(cnt):
    b = net(a)
t1 = (time.time() - t) / cnt
print(t1)
t = time.time()
for i in range(cnt):
    for a_ in a.split(a.shape[0] // div):
        b = net(a_)
t2 = (time.time() - t) / cnt
print(t2)
print(t2 / t1)
```

The first test uses batch-size 128, and the second test uses batch-size = 1 for 128 times. In our test, the first is 70-80 times faster than the second.

So how could we avoid the case of batch-size = 1? The answer is synchronize sampling: we create multiple independent environments and sample simultaneously. It is similar to A2C, but other algorithms can also use this method. In our experiments, sampling from more environments benefits not only the sample speed but also the converge speed of neural network (we guess it lowers the sample bias).

By the way, A2C is better than A3C in some cases: A3C needs to act independently and sync the gradient to master, but, in a single node, using A3C to act with batch-size = 1 is quite resource-consuming.

1.5.2 Algorithm specific tricks

Here is about the experience of hyper-parameter tuning on CartPole and Pendulum:

- *DQNPolicy*: use estimation_step = 3 or 4 and target network, also with a suitable size of replay buffer;
- *PGPolicy*: TBD
- *A2CPolicy*: TBD
- *PPOPolicy*: TBD
- *DDPGPolicy*, *TD3Policy*, and *SACPolicy*: We found two tricks. The first is to ignore the done flag. The second is to normalize reward to a standard normal distribution (it is against the theoretical analysis, but indeed works very well). The two tricks work amazingly on Mujoco tasks, typically with a faster converge speed (1M -> 200K).
- On-policy algorithms: increase the repeat-time (to 2 or 4 for trivial benchmark, 10 for mujoco) of the given batch in each training update will make the algorithm more stable.

1.5.3 Code-level optimization

Tianshou has many short-but-efficient lines of code. For example, when we want to compute $V(s)$ and $V(s')$ by the same network, the best way is to concatenate s and s' together instead of computing the value function using twice of network forward.

1.5.4 Atari/Mujoco Task Specific

Please refer to Atari examples page and Mujoco examples page.

1.5.5 Finally

With fast-speed sampling, we could use large batch-size and large learning rate for faster convergence.

RL algorithms are seed-sensitive. Try more seeds and pick the best. But for our demo, we just used seed = 0 and found it work surprisingly well on policy gradient, so we did not try other seed.

1.6 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 `gym.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*.

1.6.1 Build Policy Network

See *Build the Network*.

1.6.2 Build New Policy

See *BasePolicy*.

1.6.3 Customize Training Process

See *Train a Policy with Customized Codes*.

1.6.4 Parallel Sampling

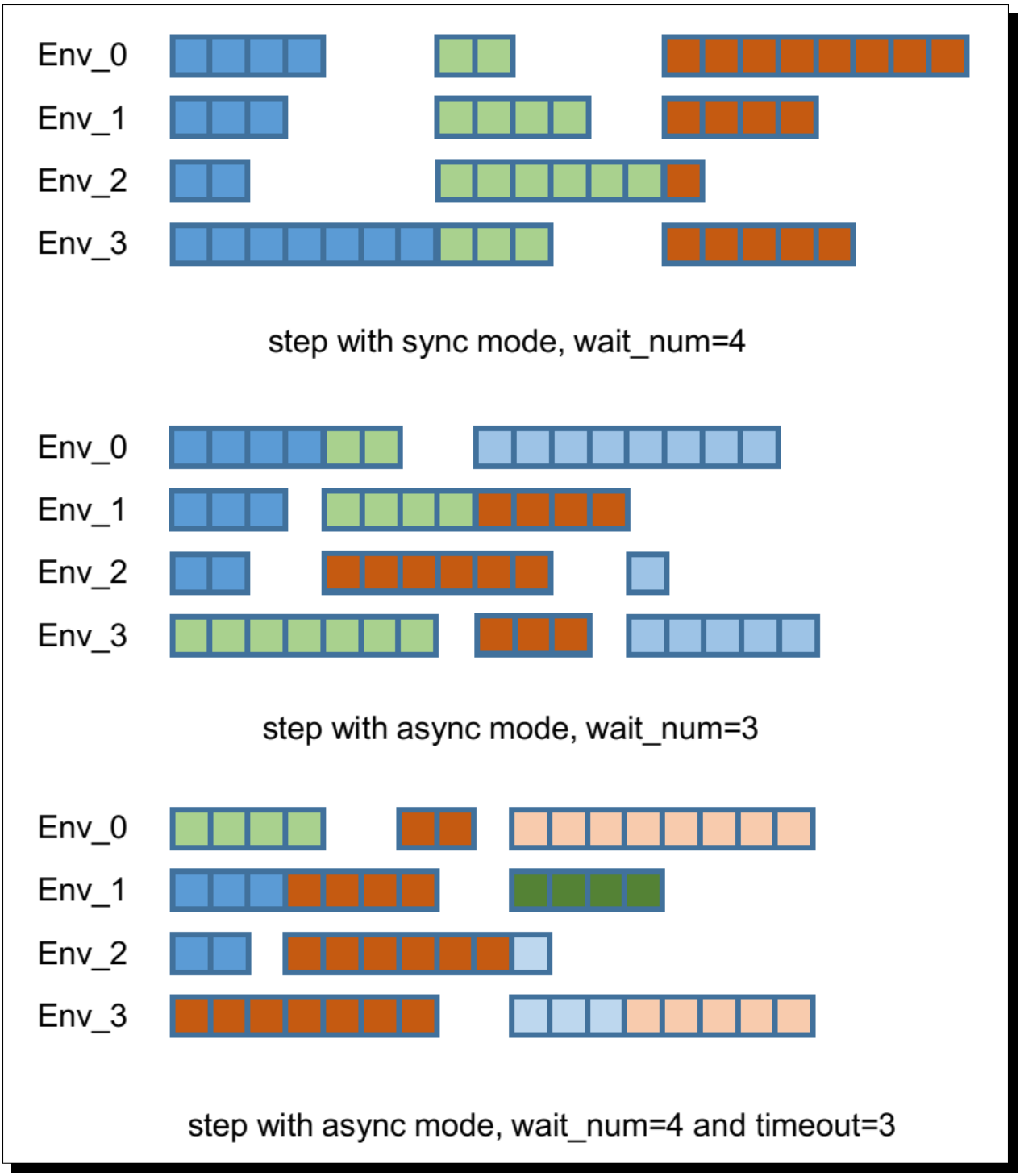
Tianshou provides the following classes for parallel environment simulation:

- *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.

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.

1.6.5 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.

This function receives up to 7 keys `obs`, `act`, `rew`, `done`, `obs_next`, `info`, and `policy`, as listed in *`Batch`*. It returns the modified part within a *`Batch`*. Only `obs` is defined at env.reset, while every key is specified for normal steps.

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 only obs exist -> reset
        # if obs/act/rew/done/... exist -> normal step
        if 'rew' not in kwargs:
            # means that it is called after env.reset(), it can only process the obs
```

(continues on next page)

(continued from previous page)

```
            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']:
                    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, test_processor.preprocess_fn)
```

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

1.6.6 RNN-style Training

This is related to Issue 19.

First, add an argument "stack_num" to *ReplayBuffer*:

```
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.

1.6.7 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 ReplayBuffer
>>> b = ReplayBuffer(size=3)
>>> b.add(obs=e.reset(), act=0, rew=0, done=0)
>>> print(b)
ReplayBuffer(
    act: array([0, 0, 0]),
    done: array([0, 0, 0]),
    info: Batch(),
    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]]),
         ),
    policy: Batch(),
    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, indice = b.sample(2)
>>> batch.keys()
['act', 'done', 'info', 'obs', 'obs_next', 'policy', 'rew']
>>> batch.obs[-1]
Batch(
```

(continues on next page)

(continued from previous page)

```
    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
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(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(a):
    ...
    return copy.deepcopy(self.graph), reward, done, {}
```

1.6.8 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:

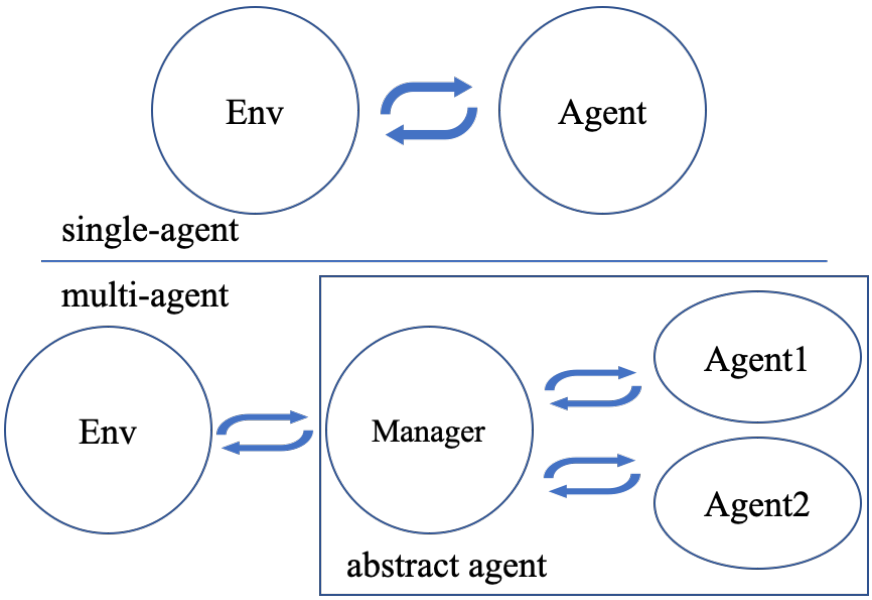
1. Simultaneous move: at each timestep, all the agents take their actions (example: moba games)
2. Cyclic move: players take action in turn (example: Go game)
3. Conditional move, at each timestep, the environment conditionally selects an agent to take action. (example: Pig Game)

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:

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

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

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

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

1.7 tianshou.data

class tianshou.data.**Batch**(*batch_dict: Optional[Union[dict, Batch, Sequence[Union[dict, Batch]], numpy.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 themself. 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, numpy.integer, numpy.ndarray, List[int]]*) → Any
Return self[index].

__len__() → int
Return len(self).

__setitem__(*index: Union[str, slice, int, numpy.integer, numpy.ndarray, List[int]]*, *value: Any*) → None
Assign value to self[index].

static cat(*batches: Sequence[Union[dict, Batch]]*) → tianshou.data.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)
```

cat_(*batches: Union[Batch, Sequence[Union[dict, Batch]]]*) → None
Concatenate a list of (or one) Batch objects into current batch.

static empty(*batch: tianshou.data.batch.Batch*, *index: Union[str, slice, int, numpy.integer, numpy.ndarray, List[int]] = None*) → tianshou.data.batch.Batch
Return an empty Batch object with 0 or None filled.

The shape is the same as the given Batch.

empty_(*index: Union[str, slice, int, numpy.integer, numpy.ndarray, List[int]] = None*) → tianshou.data.batch.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.]),
       ),
)
```

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
Return self.shape.

split(*size: int*, *shuffle: bool = True*, *merge_last: bool = False*) → Iterator[tianshou.data.batch.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.

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

stack_(*batches: Sequence[Union[dict, Batch]]*, *axis: int = 0*) → None
Stack a list of Batch object into current batch.

to_numpy() → None
Change all torch.Tensor to numpy.ndarray in-place.

to_torch(*dtype: Optional[torch.dtype] = None*, *device: Union[str, int, torch.device] = 'cpu'*) → None
Change all numpy.ndarray to torch.Tensor in-place.

update(*batch: Optional[Union[dict, Batch]] = None*, ***kwargs: Any*) → None
Update this batch from another dict/Batch.

class tianshou.data.**Collector**(*policy: tianshou.policy.base.BasePolicy*, *env: Union[gym.core.Env, tianshou.env.venvs.BaseVectorEnv]*, *buffer: Optional[tianshou.data.buffer.ReplayBuffer] = None*, *preprocess_fn: Optional[Callable[[...], tianshou.data.batch.Batch]] = None*, *action_noise: Optional[tianshou.exploration.random.BaseNoise] = None*, *reward_metric: Optional[Callable[[numpy.ndarray], float]] = None*)

Bases: `object`

Collector enables the policy to interact with different types of envs.

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` (testing phase), it will not store the data.
- **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*, defaults to None.
- **action_noise** (`BaseNoise`) – add a noise to continuous action. Normally a policy already has a noise param for exploration in training phase, so this is recommended to use in test collector for some purpose.
- **reward_metric** (*function*) – to be used in multi-agent RL. The reward to report is of shape [agent_num], but we need to return a single scalar to monitor training. This function specifies what is the desired metric, e.g., the reward of agent 1 or the average reward over all agents. By default, the behavior is to select the reward of agent 1.

The `preprocess_fn` is a function called before the data has been added to the buffer with batch format, which receives up to 7 keys as listed in *Batch*. It will receive with only `obs` when the collector resets the environment. It returns either a dict or a *Batch* with the modified keys and values. Examples are in "test/base/test_collector.py".

Here is the example:

```
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
envs = DummyVectorEnv([lambda: gym.make('CartPole-v0')
                       for _ in range(3)])
collector = Collector(policy, envs, buffer=replay_buffer)

# collect 3 episodes
collector.collect(n_episode=3)
# collect 1 episode for the first env, 3 for the third env
collector.collect(n_episode=[1, 0, 3])
# collect at least 2 steps
collector.collect(n_step=2)
# collect episodes with visual rendering (the render argument is the
#   sleep time between rendering consecutive frames)
collector.collect(n_episode=1, render=0.03)
```

Collected data always consist of full episodes. So if only `n_step` argument is give, the collector may return the data more than the `n_step` limitation. Same as `n_episode` for the multiple environment case.

Note: Please make sure the given environment has a time limitation.

---

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

Collect a specified number of step or episode.

Parameters

- **n_step** (*int*) – how many steps you want to collect.
- **n_episode** – how many episodes you want to collect. If it is an int, it means to collect at lease `n_episode` episodes; if it is a list, it means to collect exactly `n_episode[i]` episodes in the i-th environment
- **random** (*bool*) – whether to use random policy for collecting data, defaults to False.
- **render** (*float*) – the sleep time between rendering consecutive frames, defaults to None (no rendering).
- **no_grad** (*bool*) – whether to retain gradient in policy.forward, defaults to True (no gradient retaining).

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` the collected number of episodes.
- `n/st` the collected number of steps.
- `v/st` the speed of steps per second.
- `v/ep` the speed of episode per second.
- `rew` the mean reward over collected episodes.
- `len` the mean length over collected episodes.

get_env_num() → int
Return the number of environments the collector have.

reset() → None
Reset all related variables in the collector.

reset_buffer() → None
Reset the main data buffer.

reset_env() → None
Reset all of the environment(s)' states and the cache buffers.

reset_stat() → None
Reset the statistic variables.

class tianshou.data.**ListReplayBuffer**(***kwargs: Any*)
Bases: tianshou.data.buffer.ReplayBuffer

List-based replay buffer.

The function of *ListReplayBuffer* is almost the same as *ReplayBuffer*. The only difference is that *ListReplayBuffer* is based on list. Therefore, it does not support advanced indexing, which means you cannot sample a batch of data out of it. It is typically used for storing data.

See also:

Please refer to *ReplayBuffer* for more detailed explanation.

reset() → None
Clear all the data in replay buffer.

sample(*batch_size: int*) → Tuple[tianshou.data.batch.Batch, numpy.ndarray]
Get a random sample from buffer with size equal to batch_size.

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

Returns Sample data and its corresponding index inside the buffer.

class tianshou.data.**PrioritizedReplayBuffer**(*size: int*, *alpha: float*, *beta: float*, ***kwargs: Any*)
Bases: tianshou.data.buffer.ReplayBuffer

Implementation of Prioritized Experience Replay. arXiv:1511.05952.

Parameters

- **alpha** (*float*) – the prioritization exponent.
- **beta** (*float*) – the importance sample soft coefficient.

See also:

Please refer to *ReplayBuffer* for more detailed explanation.

__getitem__(*index: Union[slice, int, numpy.integer, numpy.ndarray]*) → tianshou.data.batch.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, ...).

add(*obs: Any*, *act: Any*, *rew: Union[numbers.Number, numpy.number, numpy.ndarray]*, *done: Union[numbers.Number, numpy.number, numpy.bool_]*, *obs_next: Any = None*, *info: Optional[Union[dict, tianshou.data.batch.Batch]] = {}*, *policy: Optional[Union[dict, tianshou.data.batch.Batch]] = {}*, *weight: Optional[Union[numbers.Number, numpy.number]] = None*, ***kwargs: Any*) → None
Add a batch of data into replay buffer.

sample(*batch_size: int*) → Tuple[tianshou.data.batch.Batch, numpy.ndarray]
Get a random sample from buffer with priority probability.

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

Returns Sample data and its corresponding index inside the buffer.

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

update_weight(*indice: numpy.ndarray*, *new_weight: Union[numpy.ndarray, torch.Tensor]*) → None
Update priority weight by indice in this buffer.

Parameters

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

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*)

Bases: object

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

The current implementation of Tianshou typically use 7 reserved keys in *Batch*:

- obs the observation of step t ;
- act the action of step t ;
- rew the reward of step t ;
- done the done flag of step t ;
- 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;

The following code snippet illustrates its usage:

```
>>> import pickle, numpy as np
>>> from tianshou.data import ReplayBuffer
>>> buf = ReplayBuffer(size=20)
>>> for i in range(3):
...     buf.add(obs=i, act=i, rew=i, done=i, 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.,
```

(continues on next page)

(continued from previous page)

```
       0., 0., 0., 0.])
>>> # but there are only three valid items, so len(buf) == 3.
>>> len(buf)
3
>>> pickle.dump(buf, open('buf.pkl', 'wb'))  # save to file "buf.pkl"
>>> buf2 = ReplayBuffer(size=10)
>>> for i in range(15):
...     buf2.add(obs=i, act=i, rew=i, done=i, 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)
array([ 0.,  1.,  2.,  5.,  6.,  7.,  8.,  9., 10., 11., 12., 13., 14.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.])

>>> # get a random sample from buffer
>>> # the batch_data is equal to buf[incide].
>>> batch_data, indice = buf.sample(batch_size=4)
>>> batch_data.obs == buf[indice].obs
array([ True,  True,  True,  True])
>>> len(buf)
13
>>> buf = pickle.load(open('buf.pkl', 'rb'))  # load from "buf.pkl"
>>> 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):
...     done = i % 5 == 0
...     buf.add(obs={'id': i}, act=i, rew=i, done=done,
...             obs_next={'id': i + 1})
>>> print(buf)  # you can see obs_next is not saved in buf
ReplayBuffer(
    act: array([ 9., 10., 11., 12., 13., 14., 15.,  7.,  8.]),
    done: array([0., 1., 0., 0., 0., 0., 1., 0., 0.]),
    info: Batch(),
    obs: Batch(
             id: array([ 9., 10., 11., 12., 13., 14., 15.,  7.,  8.]),
         ),
    policy: Batch(),
    rew: array([ 9., 10., 11., 12., 13., 14., 15.,  7.,  8.]),
)
>>> 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.]
```

(continues on next page)

(continued from previous page)

```
 [11. 12. 13. 14.]
 [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()
0.0
>>> # we can get obs_next through __getitem__, even if it doesn't exist
>>> print(buf[:].obs_next.id)
[[ 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.]
 [ 7.  7.  7.  8.]
 [ 7.  7.  8.  9.]]
```

Parameters

- **size** (*int*) – the size of replay buffer.
- **stack_num** (*int*) – the frame-stack sampling argument, should be greater than or equal to 1, defaults to 1 (no stacking).
- **ignore_obs_next** (*bool*) – whether to store obs_next, defaults 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, defaults to False.
- **sample_avail** (*bool*) – the parameter indicating sampling only available index when using frame-stack sampling method, defaults to False. This feature is not supported in Prioritized Replay Buffer currently.

__getitem__(*index: Union[slice, int, numpy.integer, numpy.ndarray]*) → tianshou.data.batch.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, ...).

__len__() → int
Return len(self).

add(*obs: Any*, *act: Any*, *rew: Union[numbers.Number, numpy.number, numpy.ndarray]*, *done: Union[numbers.Number, numpy.number, numpy.bool_]*, *obs_next: Any = None*, *info: Optional[Union[dict, tianshou.data.batch.Batch]] = {}*, *policy: Optional[Union[dict, tianshou.data.batch.Batch]] = {}*, ***kwargs: Any*) → None
Add a batch of data into replay buffer.

get(*indice: Union[slice, int, numpy.integer, numpy.ndarray]*, *key: str*, *stack_num: Optional[int] = None*) → Union[tianshou.data.batch.Batch, numpy.ndarray]
Return the stacked result.

E.g. [s_{t-3}, s_{t-2}, s_{t-1}, s_t], where s is self.key, t is the indice. The stack_num (here equals to 4) is given from buffer initialization procedure.

reset() → None
Clear all the data in replay buffer.

sample(*batch_size: int*) → Tuple[tianshou.data.batch.Batch, numpy.ndarray]
Get a random sample from buffer with size equal to batch_size.

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

Returns Sample data and its corresponding index inside the buffer.

property stack_num

update(*buffer: tianshou.data.buffer.ReplayBuffer*) → None
Move the data from the given buffer to self.

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.

__getitem__(*index: Union[int, numpy.ndarray]*) → Union[float, numpy.ndarray]
Return self[index].

__len__() → int

__setitem__(*index: Union[int, numpy.ndarray]*, *value: Union[float, numpy.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]
```

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

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

tianshou.data.**to_numpy**(*x: Optional[Union[tianshou.data.batch.Batch, dict, list, tuple, numpy.number, numpy.bool_, numbers.Number, numpy.ndarray, torch.Tensor]]*) → Union[tianshou.data.batch.Batch, dict, list, tuple, numpy.ndarray]
Return an object without torch.Tensor.

tianshou.data.**to_torch**(*x: Union[tianshou.data.batch.Batch, dict, list, tuple, numpy.number, numpy.bool_, numbers.Number, numpy.ndarray, torch.Tensor]*, *dtype: Optional[torch.dtype] = None*, *device: Union[str, int, torch.device] = 'cpu'*) → Union[tianshou.data.batch.Batch, dict, list, tuple, torch.Tensor]

Return an object without np.ndarray.

tianshou.data.**to_torch_as**(*x: Union[tianshou.data.batch.Batch, dict, list, tuple, numpy.ndarray, torch.Tensor]*, *y: torch.Tensor*) → Union[tianshou.data.batch.Batch, dict, list, tuple, torch.Tensor]

Return an object without np.ndarray.

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

1.8 tianshou.env

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

Bases: `gym.core.Env`

Base class for vectorized environments wrapper.

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 ith 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.

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.

render(***kwargs: Any*) → List[Any]
Render all of the environments.

reset(*id: Optional[Union[int, List[int], numpy.ndarray]] = None*) → numpy.ndarray
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.

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".

step(*action: numpy.ndarray*, *id: Optional[Union[int, List[int], numpy.ndarray]] = None*) → List[numpy.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 including four items:

- `obs` a numpy.ndarray, the agent's observation of current environments
- `rew` a numpy.ndarray, the amount of rewards returned after previous actions
- `done` a numpy.ndarray, whether these episodes have ended, in which case further step() calls will return undefined results
- `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.

class tianshou.env.**DummyVectorEnv**(*env_fns: List[Callable[], gym.core.Env]]*, *wait_num: Optional[int] = None*, *timeout: Optional[float] = None*)

Bases: `tianshou.env.venvs.BaseVectorEnv`

Dummy vectorized environment wrapper, implemented in for-loop.

See also:

Please refer to *BaseVectorEnv* for more detailed explanation.

class tianshou.env.**MultiAgentEnv**

Bases: `abc.ABC`, `gym.core.Env`

The interface for multi-agent environments.

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

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

The available action's mask is set to 1, otherwise it is set to 0. Further usage can be found at *Multi-Agent Reinforcement Learning*.

abstract reset() → dict

Reset the state.

Return the initial state, first agent_id, and the initial action set, for example, `{'obs': obs, 'agent_id': agent_id, 'mask': mask}`.

abstract step(*action: numpy.ndarray*) → Tuple[Dict[str, Any], numpy.ndarray, numpy.ndarray, numpy.ndarray]

Run one timestep of the environment's dynamics.

When the end of episode is reached, you are responsible for calling reset() to reset the environment's state.

Accept action and return a tuple (obs, rew, done, info).

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

Returns

A tuple including four items:

- `obs` a dict containing obs, agent_id, and mask, which means that it is the `agent_id` player's turn to play with `obs` observation and `mask`.
- `rew` a numpy.ndarray, the amount of rewards returned after previous actions. Depending on the specific environment, this can be either a scalar reward for current agent or a vector reward for all the agents.
- `done` a numpy.ndarray, whether the episode has ended, in which case further step() calls will return undefined results
- `info` a numpy.ndarray, contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

class tianshou.env.**RayVectorEnv**(*env_fns: List[Callable[], gym.core.Env]]*, *wait_num: Optional[int] = None*, *timeout: Optional[float] = None*)

Bases: tianshou.env.venvs.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 more detailed explanation.

class tianshou.env.**ShmemVectorEnv**(*env_fns: List[Callable[], gym.core.Env]]*, *wait_num: Optional[int] = None*, *timeout: Optional[float] = None*)

Bases: tianshou.env.venvs.BaseVectorEnv

Optimized SubprocVectorEnv with shared buffers to exchange observations.

ShmemVectorEnv has exactly the same API as SubprocVectorEnv.

See also:

Please refer to *SubprocVectorEnv* for more detailed explanation.

class tianshou.env.**SubprocVectorEnv**(*env_fns: List[Callable[], gym.core.Env]]*, *wait_num: Optional[int] = None*, *timeout: Optional[float] = None*)

Bases: tianshou.env.venvs.BaseVectorEnv

Vectorized environment wrapper based on subprocess.

See also:

Please refer to *BaseVectorEnv* for more detailed explanation.

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

Bases: tianshou.env.worker.base.EnvWorker

Dummy worker used in sequential vector environments.

close_env() → None

render(***kwargs: Any*) → Any

Render the environment.

reset() → Any

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

send_action(*action: numpy.ndarray*) → None

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

Given a list of workers, return those ready ones.

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

Bases: abc.ABC

An abstract worker for an environment.

close() → None

abstract close_env() → None

get_result() → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]

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

Render the environment.

abstract reset() → Any

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

abstract send_action(*action: numpy.ndarray*) → None

step(*action: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]
Perform one timestep of the environment's dynamic.

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

static wait(*workers: List[EnvWorker], wait_num: int, timeout: Optional[float] = None*) → List[tianshou.env.worker.base.EnvWorker]
Given a list of workers, return those ready ones.

class tianshou.env.worker.**RayEnvWorker**(*env_fn: Callable[], gym.core.Env]*)
Bases: tianshou.env.worker.base.EnvWorker

Ray worker used in RayVectorEnv.

close_env() → None

get_result() → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]

render(***kwargs: Any*) → Any
Render the environment.

reset() → Any

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

send_action(*action: numpy.ndarray*) → None

static wait(*workers: List[RayEnvWorker], wait_num: int, timeout: Optional[float] = None*) → List[tianshou.env.worker.ray.RayEnvWorker]
Given a list of workers, return those ready ones.

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

Subprocess worker used in SubprocVectorEnv and ShmemVectorEnv.

close_env() → None

get_result() → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]

render(***kwargs: Any*) → Any
Render the environment.

reset() → Any

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

send_action(*action: numpy.ndarray*) → None

static wait(*workers: List[SubprocEnvWorker], wait_num: int, timeout: Optional[float] = None*) → List[tianshou.env.worker.subproc.SubprocEnvWorker]
Given a list of workers, return those ready ones.

1.9 tianshou.policy

class tianshou.policy.**A2CPolicy**(*actor: torch.nn.modules.module.Module*, *critic: torch.nn.modules.module.Module*, *optim: torch.optim.optimizer.Optimizer*, *dist_fn: Callable[], torch.distributions.distribution.Distribution]*, *discount_factor: float = 0.99*, *vf_coef: float = 0.5*, *ent_coef: float = 0.01*, *max_grad_norm: Optional[float] = None*, *gae_lambda: float = 0.95*, *reward_normalization: bool = False*, *max_batchsize: int = 256*, ***kwargs: Any*)

Bases: tianshou.policy.modelfree.pg.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** (*Callable[[], torch.distributions.Distribution]*) – distribution class for computing the action.
- **discount_factor** (*float*) – in [0, 1], defaults to 0.99.
- **vf_coef** (*float*) – weight for value loss, defaults to 0.5.
- **ent_coef** (*float*) – weight for entropy loss, defaults to 0.01.
- **max_grad_norm** (*float*) – clipping gradients in back propagation, defaults to None.
- **gae_lambda** (*float*) – in [0, 1], param for Generalized Advantage Estimation, defaults to 0.95.
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1), defaults 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; defaults to 256.

See also:

Please refer to *BasePolicy* for more detailed explanation.

forward(*batch: tianshou.data.batch.Batch*, *state: Optional[Union[dict, tianshou.data.batch.Batch, numpy.ndarray]] = None*, ***kwargs: Any*) → tianshou.data.batch.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: tianshou.data.batch.Batch*, *batch_size: int*, *repeat: int*, ***kwargs: Any*) → Dict[str, List[float]]
Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

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.

process_fn(*batch: tianshou.data.batch.Batch*, *buffer: tianshou.data.buffer.ReplayBuffer*, *indice: numpy.ndarray*) → tianshou.data.batch.Batch
Compute the discounted returns for each frame.

$$G_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$$

where T is the terminal time step, γ is the discount factor, $\gamma \in [0, 1]$.

training: bool

class tianshou.policy.**BasePolicy**(*observation_space: gym.spaces.space.Space = None*, *action_space: gym.spaces.space.Space = None*)
Bases: `abc.ABC`, `torch.nn.modules.module.Module`

The base class for any RL policy.

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

A policy class typically has four 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.

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"))
```

static compute_episodic_return(*batch: tianshou.data.batch.Batch*, *v_s_: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *gamma: float = 0.99*, *gae_lambda: float = 0.95*, *rew_norm: bool = False*) → tianshou.data.batch.Batch

Compute returns over given full-length episodes.

Implementation of Generalized Advantage Estimator (arXiv:1506.02438).

Parameters

- **batch** (*Batch*) – a data batch which contains several full-episode data chronologically.
- **v_s** (*numpy.ndarray*) – the value function of all next states $V(s')$.
- **gamma** (*float*) – the discount factor, should be in [0, 1], defaults to 0.99.
- **gae_lambda** (*float*) – the parameter for Generalized Advantage Estimation, should be in [0, 1], defaults to 0.95.
- **rew_norm** (*bool*) – normalize the reward to Normal(0, 1), defaults to False.

Returns a Batch. The result will be stored in batch.returns as a numpy array with shape (bsz,).

static compute_nstep_return(*batch: tianshou.data.batch.Batch*, *buffer: tianshou.data.buffer.ReplayBuffer*, *indice: numpy.ndarray*, *target_q_fn: Callable[[tianshou.data.buffer.ReplayBuffer, numpy.ndarray], torch.Tensor]*, *gamma: float = 0.99*, *n_step: int = 1*, *rew_norm: bool = False*) → tianshou.data.batch.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*) – a data buffer which contains several full-episode data chronologically.
- **indice** (*numpy.ndarray*) – sampled timestep.
- **target_q_fn** (*function*) – a function receives $t + n - 1$ step's data and compute target Q value.
- **gamma** (*float*) – the discount factor, should be in [0, 1], defaults to 0.99.
- **n_step** (*int*) – the number of estimation step, should be an int greater than 0, defaults to 1.
- **rew_norm** (*bool*) – normalize the reward to Normal(0, 1), defaults to False.

Returns a Batch. The result will be stored in batch.returns as a torch.Tensor with shape (bsz,).

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

abstract learn(*batch: tianshou.data.batch.Batch*, ***kwargs: Any*) → Mapping[str, Union[float, List[float]]]

Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

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: tianshou.data.batch.Batch*, *buffer: tianshou.data.buffer.ReplayBuffer*, *indice: numpy.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()*.

process_fn(*batch: tianshou.data.batch.Batch*, *buffer: tianshou.data.buffer.ReplayBuffer*, *indice: numpy.ndarray*) → tianshou.data.batch.Batch

Pre-process the data from the provided replay buffer.

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

set_agent_id(*agent_id: int*) → None

Set self.agent_id = agent_id, for MARL.

training: bool

update(*sample_size: int*, *buffer: Optional[tianshou.data.buffer.ReplayBuffer]*, ***kwargs: Any*) → Mapping[str, Union[float, List[float]]]

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.

class tianshou.policy.**DDPGPolicy**(*actor: Optional[torch.nn.modules.module.Module]*, *actor_optim: Optional[torch.optim.optimizer.Optimizer]*, *critic: Optional[torch.nn.modules.module.Module]*, *critic_optim: Optional[torch.optim.optimizer.Optimizer]*, *action_range: Tuple[float, float]*, *tau: float = 0.005*, *gamma: float = 0.99*, *exploration_noise: Optional[tianshou.exploration.random.BaseNoise] = <tianshou.exploration.random.GaussianNoise object>*, *reward_normalization: bool = False*, *ignore_done: bool = False*, *estimation_step: int = 1*, ***kwargs: Any*)

Bases: `tianshou.policy.base.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.
- **action_range** (*Tuple[float, float]*) – the action range (minimum, maximum).
- **tau** (*float*) – param for soft update of the target network, defaults to 0.005.
- **gamma** (*float*) – discount factor, in [0, 1], defaults to 0.99.
- **exploration_noise** (BaseNoise) – the exploration noise, add to the action, defaults to `GaussianNoise(sigma=0.1)`.
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1), defaults to False.
- **ignore_done** (*bool*) – ignore the done flag while training the policy, defaults to False.
- **estimation_step** (*int*) – greater than 1, the number of steps to look ahead.

See also:

Please refer to *BasePolicy* for more detailed explanation.

forward(*batch: tianshou.data.batch.Batch*, *state: Optional[Union[dict, tianshou.data.batch.Batch, numpy.ndarray]] = None*, *model: str = 'actor'*, *input: str = 'obs'*, ***kwargs: Any*) → tianshou.data.batch.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: tianshou.data.batch.Batch*, ***kwargs: Any*) → Dict[str, float]
Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

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.

process_fn(*batch: tianshou.data.batch.Batch*, *buffer: tianshou.data.buffer.ReplayBuffer*, *indice: numpy.ndarray*) → tianshou.data.batch.Batch
Pre-process the data from the provided replay buffer.

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

set_exp_noise(*noise: Optional[tianshou.exploration.random.BaseNoise]*) → None
Set the exploration noise.

sync_weight() → None
Soft-update the weight for the target network.

train(*mode: bool = True*) → tianshou.policy.modelfree.ddpg.DDPGPolicy
Set the module in training mode, except for the target network.

training: bool

class tianshou.policy.**DQNPolicy**(*model: torch.nn.modules.module.Module*, *optim: torch.optim.optimizer.Optimizer*, *discount_factor: float = 0.99*, *estimation_step: int = 1*, *target_update_freq: int = 0*, *reward_normalization: bool = False*, ***kwargs: Any*)
Bases: `tianshou.policy.base.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*) – greater than 1, the number of steps to look ahead.
- **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), defaults to False.

See also:

Please refer to *BasePolicy* for more detailed explanation.

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

Parameters **eps** (*float*) – in [0, 1], for epsilon-greedy exploration method.

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: tianshou.data.batch.Batch*, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

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.

process_fn(*batch: tianshou.data.batch.Batch*, *buffer: tianshou.data.buffer.ReplayBuffer*, *indice: numpy.ndarray*) → tianshou.data.batch.Batch
Compute the n-step return for Q-learning targets.

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

set_eps(*eps: float*) → None
Set the eps for epsilon-greedy exploration.

sync_weight() → None
Synchronize the weight for the target network.

train(*mode: bool = True*) → tianshou.policy.modelfree.dqn.DQNPolicy
Set the module in training mode, except for the target network.

training: bool

class tianshou.policy.**DiscreteSACPolicy**(*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*, *alpha: Union[float, Tuple[float, torch.Tensor, torch.optim.optimizer.Optimizer]] = 0.2*, *reward_normalization: bool = False*, *ignore_done: bool = False*, *estimation_step: int = 1*, ***kwargs: Any*)
Bases: tianshou.policy.modelfree.sac.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, defaults to 0.005.
- **gamma** (*float*) – discount factor, in [0, 1], defaults to 0.99.

- **torch.Tensor, torch.optim.Optimizer) or float alpha** (*(float,)*) – entropy regularization coefficient, default to 0.2. If a tuple (target_entropy, log_alpha, alpha_optim) is provided, then alpha is automatatically tuned.
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1), defaults to `False`.
- **ignore_done** (*bool*) – ignore the done flag while training the policy, defaults to `False`.

See also:

Please refer to *BasePolicy* for more detailed explanation.

forward(*batch: tianshou.data.batch.Batch*, *state: Optional[Union[dict, tianshou.data.batch.Batch, numpy.ndarray]] = None*, *input: str = 'obs'*, ***kwargs: Any*) → tianshou.data.batch.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: tianshou.data.batch.Batch*, ***kwargs: Any*) → Dict[str, float]

Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

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.**ImitationPolicy**(*model: torch.nn.modules.module.Module*, *optim: torch.optim.optimizer.Optimizer*, *mode: str = 'continuous'*, ***kwargs: Any*)

Bases: `tianshou.policy.base.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.
- **mode** (*str*) – indicate the imitation type ("continuous" or "discrete" action space), defaults to "continuous".

See also:

Please refer to `BasePolicy` for more detailed explanation.

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

learn(*batch: tianshou.data.batch.Batch*, ***kwargs: Any*) → Dict[str, float]
Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

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.**MultiAgentPolicyManager**(*policies: List[tianshou.policy.base.BasePolicy]*, ***kwargs: Any*)

Bases: `tianshou.policy.base.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.

forward(*batch: tianshou.data.batch.Batch*, *state: Optional[Union[dict, tianshou.data.batch.Batch]] = None*, ***kwargs: Any*) → tianshou.data.batch.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: tianshou.data.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
}
```

process_fn(*batch: tianshou.data.batch.Batch*, *buffer: tianshou.data.buffer.ReplayBuffer*, *indice: numpy.ndarray*) → tianshou.data.batch.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.

replace_policy(*policy: tianshou.policy.base.BasePolicy*, *agent_id: int*) → None
Replace the "agent_id"th policy in this manager.

training: bool

class tianshou.policy.**PGPolicy**(*model: Optional[torch.nn.modules.module.Module]*, *optim: torch.optim.optimizer.Optimizer*, *dist_fn: Callable[], torch.distributions.distribution.Distribution]*, *discount_factor: float = 0.99*, *reward_normalization: bool = False*, ***kwargs: Any*)
Bases: tianshou.policy.base.BasePolicy

Implementation of Vanilla Policy Gradient.

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** (*Callable[[], torch.distributions.Distribution]*) – distribution class for computing the action.

- **discount_factor** (*float*) – in [0, 1].

See also:

Please refer to *BasePolicy* for more detailed explanation.

forward(*batch: tianshou.data.batch.Batch*, *state: Optional[Union[dict, tianshou.data.batch.Batch, numpy.ndarray]] = None*, ***kwargs: Any*) → tianshou.data.batch.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: tianshou.data.batch.Batch*, *batch_size: int*, *repeat: int*, ***kwargs: Any*) → Dict[str, List[float]]
Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

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.

process_fn(*batch: tianshou.data.batch.Batch*, *buffer: tianshou.data.buffer.ReplayBuffer*, *indice: numpy.ndarray*) → tianshou.data.batch.Batch
Compute the discounted returns for each frame.

$$G_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$$

where T is the terminal time step, γ is the discount factor, $\gamma \in [0, 1]$.

training: bool

class tianshou.policy.**PPOPolicy**(*actor: torch.nn.modules.module.Module*, *critic: torch.nn.modules.module.Module*, *optim: torch.optim.optimizer.Optimizer*, *dist_fn: Callable[], torch.distributions.distribution.Distribution]*, *discount_factor: float = 0.99*, *max_grad_norm: Optional[float] = None*, *eps_clip: float = 0.2*, *vf_coef: float = 0.5*, *ent_coef: float = 0.01*, *action_range: Optional[Tuple[float, float]] = None*, *gae_lambda: float = 0.95*, *dual_clip: Optional[float] = None*, *value_clip: bool = True*, *reward_normalization: bool = True*, *max_batchsize: int = 256*, ***kwargs: Any*)

Bases: tianshou.policy.modelfree.pg.PGPolicy

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** (*Callable[[], torch.distributions.Distribution]*) – distribution class for computing the action.
- **discount_factor** (*float*) – in [0, 1], defaults to 0.99.
- **max_grad_norm** (*float*) – clipping gradients in back propagation, defaults to None.
- **eps_clip** (*float*) – ϵ in L_{CLIP} in the original paper, defaults to 0.2.
- **vf_coef** (*float*) – weight for value loss, defaults to 0.5.
- **ent_coef** (*float*) – weight for entropy loss, defaults to 0.01.
- **action_range** (*(float, float)*) – the action range (minimum, maximum).
- **gae_lambda** (*float*) – in [0, 1], param for Generalized Advantage Estimation, defaults to 0.95.
- **dual_clip** (*float*) – a parameter c mentioned in arXiv:1912.09729 Equ. 5, where c > 1 is a constant indicating the lower bound, defaults 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, defaults to True.
- **reward_normalization** (*bool*) – normalize the returns to Normal(0, 1), defaults to True.
- **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; defaults to 256.

See also:

Please refer to *BasePolicy* for more detailed explanation.

forward(*batch: tianshou.data.batch.Batch*, *state: Optional[Union[dict, tianshou.data.batch.Batch, numpy.ndarray]] = None*, ***kwargs: Any*) → tianshou.data.batch.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: tianshou.data.batch.Batch*, *batch_size: int*, *repeat: int*, ***kwargs: Any*) → Dict[str, List[float]]

Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

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.

process_fn (*batch: tianshou.data.batch.Batch*, *buffer: tianshou.data.buffer.ReplayBuffer*, *indice: numpy.ndarray*) → tianshou.data.batch.Batch

Compute the discounted returns for each frame.

$$G_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$$

where T is the terminal time step, γ is the discount factor, $\gamma \in [0, 1]$.

training: bool

class tianshou.policy.**PSRLPolicy** (*trans_count_prior: numpy.ndarray*, *rew_mean_prior: numpy.ndarray*, *rew_std_prior: numpy.ndarray*, *discount_factor: float = 0.99*, *epsilon: float = 0.01*, *add_done_loop: bool = False*, ***kwargs: Any*)

Bases: `tianshou.policy.base.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, defaults to False.

See also:

Please refer to *BasePolicy* for more detailed explanation.

forward(*batch: tianshou.data.batch.Batch*, *state: Optional[Union[dict, tianshou.data.batch.Batch, numpy.ndarray]] = None*, ***kwargs: Any*) → tianshou.data.batch.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: tianshou.data.batch.Batch*, **args: Any*, ***kwargs: Any*) → Dict[str, float]
Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

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.**RandomPolicy**(*observation_space: gym.spaces.space.Space = None*, *action_space: gym.spaces.space.Space = None*)
Bases: `tianshou.policy.base.BasePolicy`

A random agent used in multi-agent learning.

It randomly chooses an action from the legal action.

forward(*batch: tianshou.data.batch.Batch*, *state: Optional[Union[dict, tianshou.data.batch.Batch, numpy.ndarray]] = None*, ***kwargs: Any*) → tianshou.data.batch.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: tianshou.data.batch.Batch*, ***kwargs: Any*) → Dict[str, float]
Since a random agent learn nothing, it returns an empty dict.

training: bool

class tianshou.policy.**SACPolicy**(*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*, *action_range: Tuple[float, float]*, *tau: float = 0.005*, *gamma: float = 0.99*, *alpha: Union[float, Tuple[float, torch.Tensor, torch.optim.optimizer.Optimizer]] = 0.2*, *reward_normalization: bool = False*, *ignore_done: bool = False*, *estimation_step: int = 1*, *exploration_noise: Optional[tianshou.exploration.random.BaseNoise] = None*, ***kwargs: Any*)

Bases: tianshou.policy.modelfree.ddpg.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.
- **action_range** (*Tuple[float, float]*) – the action range (minimum, maximum).
- **tau** (*float*) – param for soft update of the target network, defaults to 0.005.
- **gamma** (*float*) – discount factor, in [0, 1], defaults to 0.99.
- **torch.Tensor, torch.optim.Optimizer) or float alpha** (*(float,)* – entropy regularization coefficient, default to 0.2. If a tuple (target_entropy, log_alpha, alpha_optim) is provided, then alpha is automatatically tuned.
- **reward_normalization** (*bool*) – normalize the reward to Normal(0, 1), defaults to False.
- **ignore_done** (*bool*) – ignore the done flag while training the policy, defaults to False.
- **exploration_noise** (BaseNoise) – add a noise to action for exploration, defaults to None. This is useful when solving hard-exploration problem.

See also:

Please refer to *BasePolicy* for more detailed explanation.

forward(*batch: tianshou.data.batch.Batch*, *state: Optional[Union[dict, tianshou.data.batch.Batch, numpy.ndarray]] = None*, *input: str = 'obs'*, ***kwargs: Any*) → tianshou.data.batch.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: tianshou.data.batch.Batch*, ***kwargs: Any*) → Dict[str, float]
Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

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.

sync_weight() → None
Soft-update the weight for the target network.

train(*mode: bool = True*) → tianshou.policy.modelfree.sac.SACPolicy
Set the module in training mode, except for the target network.

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, action_range: Tuple[float, float], tau: float = 0.005, gamma: float = 0.99, exploration_noise: 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, ignore_done: bool = False, estimation_step: int = 1, **kwargs: Any*)
Bases: `tianshou.policy.modelfree.ddpg.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.
- **action_range** (*Tuple[float, float]*) – the action range (minimum, maximum).
- **tau** (*float*) – param for soft update of the target network, defaults to 0.005.
- **gamma** (*float*) – discount factor, in [0, 1], defaults to 0.99.
- **exploration_noise** (*float*) – the exploration noise, add to the action, defaults 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), defaults to False.
- **ignore_done** (*bool*) – ignore the done flag while training the policy, defaults to False.

See also:

Please refer to *BasePolicy* for more detailed explanation.

learn(*batch: tianshou.data.batch.Batch*, ***kwargs: Any*) → Dict[str, float]
Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

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.

sync_weight() → None
Soft-update the weight for the target network.

train(*mode: bool = True*) → tianshou.policy.modelfree.td3.TD3Policy
Set the module in training mode, except for the target network.

training: bool

1.10 tianshou.trainer

tianshou.trainer.**gather_info**(*start_time: float*, *train_c: tianshou.data.collector.Collector*, *test_c: tianshou.data.collector.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 frames in the training collector;
- `train_time/model` the time for training models;
- `train_speed` the speed of training (frames 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 (frames per second);
- `best_reward` the best reward over the test results;
- `duration` the total elapsed time.

tianshou.trainer.**offpolicy_trainer**(*policy: tianshou.policy.base.BasePolicy*, *train_collector: tianshou.data.collector.Collector*, *test_collector: tianshou.data.collector.Collector*, *max_epoch: int*, *step_per_epoch: int*, *collect_per_step: int*, *episode_per_test: Union[int, List[int]]*, *batch_size: int*, *update_per_step: int = 1*, *train_fn: Optional[Callable[[int, int], None]] = None*, *test_fn: Optional[Callable[[int, Optional[int]], None]] = None*, *stop_fn: Optional[Callable[[float], bool]] = None*, *save_fn: Optional[Callable[[tianshou.policy.base.BasePolicy], None]] = None*, *writer: Optional[torch.utils.tensorboard.writer.SummaryWriter] = None*, *log_interval: int = 1*, *verbose: bool = True*, *test_in_train: bool = True*) → Dict[str, Union[float, str]]

A wrapper for off-policy trainer procedure.

The "step" in trainer means a policy network update.

Parameters

- **policy** – an instance of the *BasePolicy* class.
- **train_collector** (*Collector*) – the collector used for training.
- **test_collector** (*Collector*) – the collector used for testing.
- **max_epoch** (*int*) – the maximum of epochs for training. The training process might be finished before reaching the `max_epoch`.
- **step_per_epoch** (*int*) – the number of step for updating policy network in one epoch.

- **collect_per_step** (*int*) – the number of frames the collector would collect before the network update. In other words, collect some frames and do some policy network update.
- **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*) – the number of times the policy network would be updated after frames are collected, for example, set it to 256 means it updates policy 256 times once after `collect_per_step` frames are collected.
- **train_fn** (*function*) – a function receives the current number of epoch and step index, and performs some operations at the beginning of training in this epoch.
- **test_fn** (*function*) – a function receives the current number of epoch and step index, and performs some operations at the beginning of testing in this epoch.
- **save_fn** (*function*) – a function for saving policy when the undiscounted average mean reward in evaluation phase gets better.
- **stop_fn** (*function*) – a function receives the average undiscounted returns of the testing result, return a boolean which indicates whether reaching the goal.
- **writer** (*torch.utils.tensorboard.SummaryWriter*) – a TensorBoard SummaryWriter.
- **log_interval** (*int*) – the log interval of the writer.
- **verbose** (*bool*) – whether to print the information.
- **test_in_train** (*bool*) – whether to test in the training phase.

Returns See *gather_info()*.

tianshou.trainer.**onpolicy_trainer**(*policy: tianshou.policy.base.BasePolicy*, *train_collector: tianshou.data.collector.Collector*, *test_collector: tianshou.data.collector.Collector*, *max_epoch: int*, *step_per_epoch: int*, *collect_per_step: int*, *repeat_per_collect: int*, *episode_per_test: Union[int, List[int]]*, *batch_size: int*, *train_fn: Optional[Callable[[int, int], None]] = None*, *test_fn: Optional[Callable[[int, Optional[int]], None]] = None*, *stop_fn: Optional[Callable[[float], bool]] = None*, *save_fn: Optional[Callable[[tianshou.policy.base.BasePolicy], None]] = None*, *writer: Optional[torch.utils.tensorboard.writer.SummaryWriter] = None*, *log_interval: int = 1*, *verbose: bool = True*, *test_in_train: bool = True*) → Dict[str, Union[float, str]]

A wrapper for on-policy trainer procedure.

The "step" in trainer means a policy network update.

Parameters

- **policy** – an instance of the *BasePolicy* class.
- **train_collector** (*Collector*) – the collector used for training.
- **test_collector** (*Collector*) – the collector used for testing.
- **max_epoch** (*int*) – the maximum of epochs for training. The training process might be finished before reaching the `max_epoch`.

- **step_per_epoch** (*int*) – the number of step for updating policy network in one epoch.
- **collect_per_step** (*int*) – the number of episodes the collector would collect before the network update. In other words, collect some episodes and do one policy network update.
- **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 or list of ints*) – 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.
- **train_fn** (*function*) – a function receives the current number of epoch and step index, and performs some operations at the beginning of training in this poch.
- **test_fn** (*function*) – a function receives the current number of epoch and step index, and performs some operations at the beginning of testing in this epoch.
- **save_fn** (*function*) – a function for saving policy when the undiscounted average mean reward in evaluation phase gets better.
- **stop_fn** (*function*) – a function receives the average undiscounted returns of the testing result, return a boolean which indicates whether reaching the goal.
- **writer** (*torch.utils.tensorboard.SummaryWriter*) – a TensorBoard SummaryWriter.
- **log_interval** (*int*) – the log interval of the writer.
- **verbose** (*bool*) – whether to print the information.
- **test_in_train** (*bool*) – whether to test in the training phase.

Returns See *gather_info()*.

tianshou.trainer.**test_episode**(*policy: tianshou.policy.base.BasePolicy*, *collector: tianshou.data.collector.Collector*, *test_fn: Optional[Callable[[int, Optional[int]], None]]*, *epoch: int*, *n_episode: Union[int, List[int]]*, *writer: Optional[torch.utils.tensorboard.writer.SummaryWriter] = None*, *global_step: Optional[int] = None*) → Dict[str, float]

A simple wrapper of testing policy in collector.

1.11 tianshou.exploration

class tianshou.exploration.**BaseNoise**

Bases: abc.ABC, object

The action noise base class.

abstract __call__(*size: Sequence[int]*) → numpy.ndarray

Generate new noise.

reset() → None

Reset to the initial state.

class tianshou.exploration.**GaussianNoise**(*mu: float = 0.0*, *sigma: float = 1.0*)

Bases: tianshou.exploration.random.BaseNoise

The vanilla gaussian process, for exploration in DDPG by default.

__call__(*size: Sequence[int]*) → numpy.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, numpy.ndarray]] = None*)
Bases: tianshou.exploration.random.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.

__call__(*size: Sequence[int]*, *mu: Optional[float] = None*) → numpy.ndarray
Generate new noise.

Return an numpy array which size is equal to `size`.

reset() → None
Reset to the initial state.

1.12 tianshou.utils

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(*x: Union[numbers.Number, numpy.number, list, numpy.ndarray, torch.Tensor]*) → numpy.number
Add a scalar into *MovAvg*.

You can add `torch.Tensor` with only one element, a python scalar, or a list of python scalar.

get() → numpy.number
Get the average.

mean() → numpy.number
Get the average. Same as *get()*.

std() → numpy.number
Get the standard deviation.

class tianshou.utils.net.common.**Net**(*layer_num: int*, *state_shape: tuple*, *action_shape: Optional[Union[tuple, int]] = 0*, *device: Union[str, int, torch.device] = 'cpu'*, *softmax: bool = False*, *concat: bool = False*, *hidden_layer_size: int = 128*, *dueling: Optional[Tuple[int, int]] = None*, *norm_layer: Optional[Callable[[int], torch.nn.modules.module.Module]] = None*)
Bases: torch.nn.modules.module.Module

Simple MLP backbone.

For advanced usage (how to customize the network), please refer to *Build the Network*.

Parameters

- **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.
- **dueling** (*bool*) – whether to use dueling network to calculate Q values (for Dueling DQN), defaults to False.
- **norm_layer** – use which normalization before ReLU, e.g., nn.LayerNorm and nn.BatchNorm1d, defaults to None.

forward(*s: Union[numpy.ndarray, torch.Tensor]*, *state: Optional[Any] = None*, *info: Dict[str, Any] = {}*) → Tuple[torch.Tensor, Any]
Mapping: s -> flatten -> logits.

training: bool

class tianshou.utils.net.common.**Recurrent**(*layer_num: int*, *state_shape: Sequence[int]*, *action_shape: Sequence[int]*, *device: Union[str, int, torch.device] = 'cpu'*, *hidden_layer_size: int = 128*)
Bases: torch.nn.modules.module.Module

Simple Recurrent network based on LSTM.

For advanced usage (how to customize the network), please refer to *Build the Network*.

forward(*s: Union[numpy.ndarray, torch.Tensor]*, *state: Optional[Dict[str, torch.Tensor]] = None*, *info: Dict[str, Any] = {}*) → Tuple[torch.Tensor, Dict[str, torch.Tensor]]
Mapping: s -> flatten -> logits.

In the evaluation mode, s should be with shape [bsz, dim]; in the training mode, s should be with shape [bsz, len, dim]. See the code and comment for more detail.

training: bool

tianshou.utils.net.common.**miniblock**(*inp: int*, *oup: int*, *norm_layer: Optional[Callable[[int], torch.nn.modules.module.Module]]*) → List[torch.nn.modules.module.Module]
Construct a miniblock with given input/output-size and norm layer.

class tianshou.utils.net.discrete.**Actor**(*preprocess_net: torch.nn.modules.module.Module*, *action_shape: Sequence[int]*, *hidden_layer_size: int = 128*, *softmax_output: bool = True*)

Bases: torch.nn.modules.module.Module

Simple actor network with MLP.

For advanced usage (how to customize the network), please refer to *Build the Network*.

forward(*s: Union[numpy.ndarray, torch.Tensor]*, *state: Optional[Any] = None*, *info: Dict[str, Any] = {}*) → Tuple[torch.Tensor, Any]
Mapping: s -> Q(s, *).

training: bool

class tianshou.utils.net.discrete.**Critic**(*preprocess_net: torch.nn.modules.module.Module*, *hidden_layer_size: int = 128*, *last_size: int = 1*)

Bases: torch.nn.modules.module.Module

Simple critic network with MLP.

For advanced usage (how to customize the network), please refer to *Build the Network*.

forward(*s: Union[numpy.ndarray, torch.Tensor]*, ***kwargs: Any*) → torch.Tensor
Mapping: s -> V(s).

training: bool

class tianshou.utils.net.discrete.**DQN**(*c: int*, *h: int*, *w: int*, *action_shape: Sequence[int]*, *device: Union[str, int, torch.device] = 'cpu'*)

Bases: torch.nn.modules.module.Module

Reference: Human-level control through deep reinforcement learning.

For advanced usage (how to customize the network), please refer to *Build the Network*.

forward(*x: Union[numpy.ndarray, torch.Tensor]*, *state: Optional[Any] = None*, *info: Dict[str, Any] = {}*) → Tuple[torch.Tensor, Any]
Mapping: x -> Q(x, *).

training: bool

class tianshou.utils.net.continuous.**Actor**(*preprocess_net: torch.nn.modules.module.Module*, *action_shape: Sequence[int]*, *max_action: float = 1.0*, *device: Union[str, int, torch.device] = 'cpu'*, *hidden_layer_size: int = 128*)

Bases: torch.nn.modules.module.Module

Simple actor network with MLP.

For advanced usage (how to customize the network), please refer to *Build the Network*.

forward(*s: Union[numpy.ndarray, torch.Tensor]*, *state: Optional[Any] = None*, *info: Dict[str, Any] = {}*) → Tuple[torch.Tensor, Any]
Mapping: s -> logits -> action.

training: bool

class tianshou.utils.net.continuous.**ActorProb**(*preprocess_net: torch.nn.modules.module.Module*, *action_shape: Sequence[int]*, *max_action: float = 1.0*, *device: Union[str, int, torch.device] = 'cpu'*, *unbounded: bool = False*, *hidden_layer_size: int = 128*)

Bases: torch.nn.modules.module.Module

Simple actor network (output with a Gauss distribution) with MLP.

For advanced usage (how to customize the network), please refer to *Build the Network*.

forward(*s: Union[numpy.ndarray, torch.Tensor]*, *state: Optional[Any] = None*, *info: Dict[str, Any] = {}*) → Tuple[Tuple[torch.Tensor, torch.Tensor], Any]
Mapping: s -> logits -> (mu, sigma).

training: bool

class tianshou.utils.net.continuous.**Critic**(*preprocess_net: torch.nn.modules.module.Module*, *device: Union[str, int, torch.device] = 'cpu'*, *hidden_layer_size: int = 128*)

Bases: torch.nn.modules.module.Module

Simple critic network with MLP.

For advanced usage (how to customize the network), please refer to *Build the Network*.

forward(*s: Union[numpy.ndarray, torch.Tensor]*, *a: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *info: Dict[str, Any] = {}*) → torch.Tensor
Mapping: (s, a) -> logits -> Q(s, a).

training: bool

class tianshou.utils.net.continuous.**RecurrentActorProb**(*layer_num: int*, *state_shape: Sequence[int]*, *action_shape: Sequence[int]*, *max_action: float = 1.0*, *device: Union[str, int, torch.device] = 'cpu'*, *unbounded: bool = False*, *hidden_layer_size: int = 128*)

Bases: torch.nn.modules.module.Module

Recurrent version of ActorProb.

For advanced usage (how to customize the network), please refer to *Build the Network*.

forward(*s: Union[numpy.ndarray, torch.Tensor]*, *state: Optional[Dict[str, torch.Tensor]] = None*, *info: Dict[str, Any] = {}*) → Tuple[Tuple[torch.Tensor, torch.Tensor], Dict[str, torch.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, torch.device] = 'cpu'*, *hidden_layer_size: int = 128*)

Bases: torch.nn.modules.module.Module

Recurrent version of Critic.

For advanced usage (how to customize the network), please refer to *Build the Network*.

forward(*s: Union[numpy.ndarray, torch.Tensor]*, *a: Optional[Union[numpy.ndarray, torch.Tensor]] = None*, *info: Dict[str, Any] = {}*) → torch.Tensor
Almost the same as *Recurrent*.

training: bool

1.13 Contributing to Tianshou

1.13.1 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
```

1.13.2 PEP8 Code Style Check

We follow PEP8 python code style. To check, in the main directory, run:

```
$ flake8 . --count --show-source --statistics
```

1.13.3 Type Check

We use mypy to check the type annotations. To check, in the main directory, run:

```
$ mypy
```

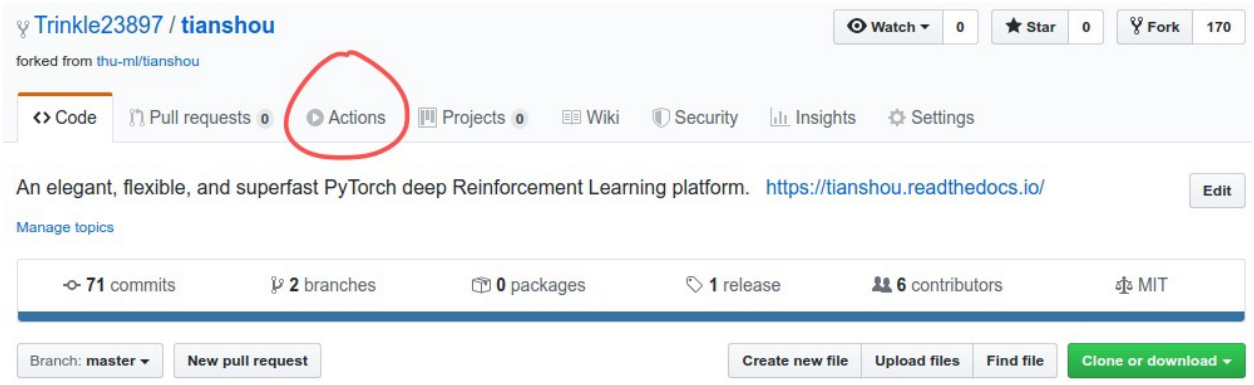
1.13.4 Test Locally

This command will run automatic tests in the main directory

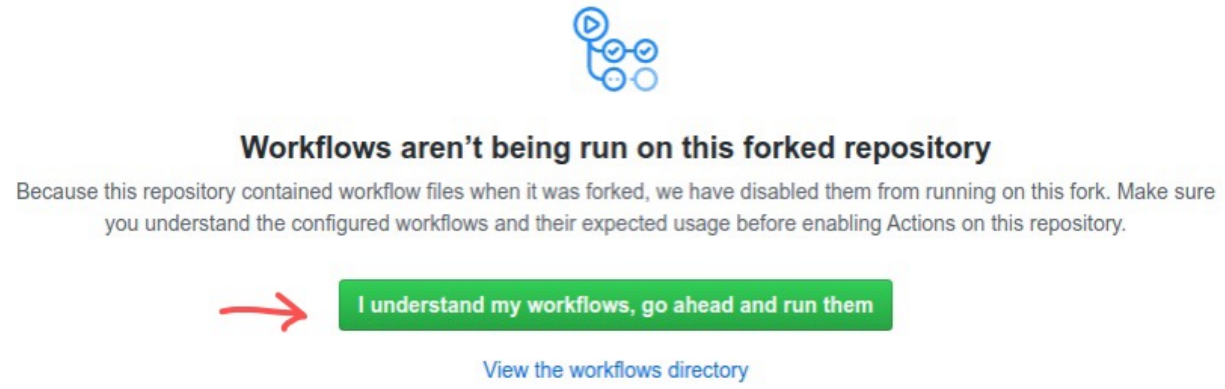
```
$ pytest test --cov tianshou -s --durations 0 -v
```

1.13.5 Test by GitHub Actions

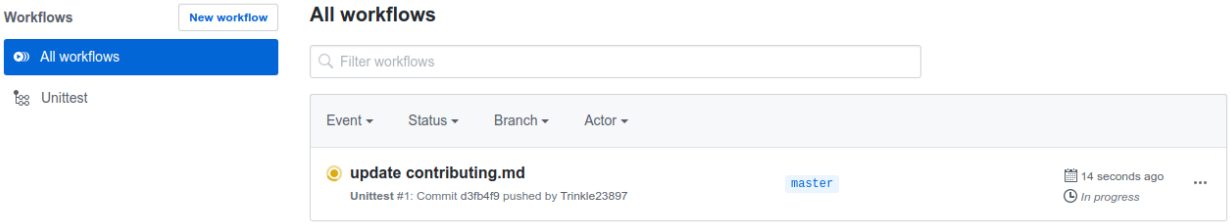
1. Click the `Actions` button in your own repo:



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:



1.13.6 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 webpages, run

```
$ make html
```

under the `docs/` directory. The generated webpages are in `docs/_build` and can be viewed with browsers.

Chinese documentation is in https://tianshou.readthedocs.io/zh/latest/.

1.13.7 Documentation Generation Test

We have the following three documentation tests:

1. pydocstyle: test docstrings under `tianshou/`. To check, in the main directory, run:

```
$ pydocstyle tianshou
```

2. doc8: test ReStructuredText formats. To check, in the main directory, run:

```
$ doc8 docs
```

3. sphinx test: test if there is any errors/warnings when generating front-end html documentations. To check, in the main directory, run:

```
$ cd docs
$ make html SPHINXOPTS="-W"
```

1.14 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)
- Minghao Zhang (Mehooz)
- Alexis Duburcq (duburcqa)
- Kaichao You (youkaichao)

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

I

L

M

N

O

P

R

S

T

U

W