Tianshou

Release 0.2.1

Tianshou contributors

Apr 10, 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 with n-step returns
- *A2CPolicy* Advantage Actor-Critic
- *DDPGPolicy* Deep Deterministic Policy Gradient
- *PPOPolicy* Proximal Policy Optimization
- *TD3Policy* Twin Delayed DDPG
- *SACPolicy* Soft Actor-Critic

Tianshou supports parallel workers for all algorithms as well. All of these algorithms are reformatted as replay-buffer based algorithms.

CHAPTER

ONE

INSTALLATION

Tianshou is currently hosted on PyPI. You can simply install Tianshou with the following command:

```
pip3 install tianshou -U
```

You can also install with the newest version through GitHub:

```
pip3 install git+https://github.com/thu-ml/tianshou.git@master
```

After installation, open your python console and type

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

If no error occurs, you have successfully installed Tianshou.

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

It is available if you want 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 three types of vectorized environment wrapper: *VectorEnv*, *SubprocVectorEnv*, and *RayVectorEnv*. It can be used as follows:

```
train_envs = ts.env.VectorEnv([lambda: gym.make('CartPole-v0') for _ in range(8)])
test_envs = ts.env.VectorEnv([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 block of codes.

1.1.3 Build the Network

Tianshou supports any user-defined PyTorch networks and optimizers but with the limitation of input and output API. Here is an example code:

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

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

The rules of self-defined networks are:

1. Input: observation `obs` (may be a `numpy.ndarray` or `torch.Tensor`), hidden state `state` (for RNN usage), and other information `info` provided by the environment.
2. Output: some `logits` and the next hidden state `state`. 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 [SWD+17], the return of the network might be `(mu, sigma), state` for Gaussian policy.

1.1.4 Setup Policy

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

```
policy = ts.policy.DQNPolicy(net, optim,
    discount_factor=0.9, estimation_step=3,
    use_target_network=True, 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 e: policy.set_eps(0.1),
    test_fn=lambda e: policy.set_eps(0.05),
    stop_fn=lambda x: x >= env.spec.reward_threshold,
    writer=None)
print(f'Finished training! Use {result["duration"]}')
```

The meaning of each parameter is as follows:

- `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 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 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',
    '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 `torch.nn.Module` class, 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:

```
collector = ts.data.Collector(policy, env)
collector.collect(n_episode=1, render=1 / 35)
collector.close()
```

1.1.9 Train a Policy with Customized Codes

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

No problem! Tianshou supports user-defined training code. Here is the usage:

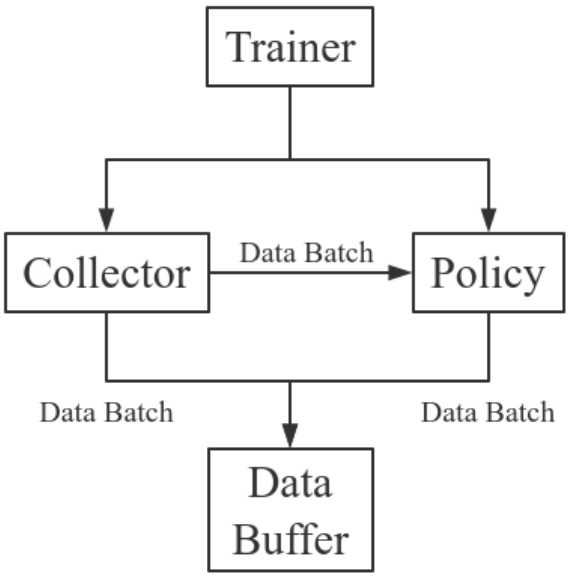
```
# pre-collect 5000 frames with random action before training
policy.set_eps(1)
train_collector.collect(n_step=5000)

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
    losses = policy.learn(train_collector.sample(batch_size=64))
```

For further usage, you can refer to *Tabular Q Learning Implementation*.

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:



1.2.1 Data 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. Here is the usage:

```
>>> import numpy as np
>>> from tianshou.data import Batch
>>> data = Batch(a=4, b=[5, 5], c='2312312')
>>> data.b
[5, 5]
>>> data.b = np.array([3, 4, 5])
>>> len(data.b)
3
>>> data.b[-1]
5
```

In short, you can define a `Batch` with any key-value pair. The current implementation of Tianshou typically use 6 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 return 4 arguments, and the last one is `info`);

Batch has other methods, including *__getitem__()*, *__len__()*, *append()*, and *split()*:

```
>>> data = Batch(obs=np.array([0, 11, 22]), rew=np.array([6, 6, 6]))
>>> # here we test __getitem__
>>> index = [2, 1]
>>> data[index].obs
```

(continues on next page)

(continued from previous page)

```
array([22, 11])

>>> # here we test __len__
>>> len(data)
3

>>> data.append(data)  # similar to list.append
>>> data.obs
array([0, 11, 22, 0, 11, 22])

>>> # split whole data into multiple small batch
>>> for d in data.split(size=2, permute=False):
...     print(d.obs, d.rew)
[ 0 11] [6 6]
[22  0] [6 6]
[11 22] [6 6]
```

1.2.2 Data Buffer

ReplayBuffer stores data generated from interaction between the policy and environment. It stores basically 6 types of data, as mentioned in *Batch*, based on `numpy.ndarray`. Here is the usage:

```
>>> 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={})
>>> len(buf)
3
>>> 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.])

>>> 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])
```

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 four parts:

- `__init__()`: initialize the policy, including coping the target network and so on;
- *__call__()*: 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.

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 \
```

(continues on next page)

(continued from previous page)

```
            + 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. In short, *Collector* has two main methods:

- *collect()*: 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;
- *sample()*: sample a data batch from replay buffer; it will call *process_fn()* before returning the final batch data.

Why do we mention **at least** here? For a single environment, the collector will finish exactly `n_step` or `n_episode`. However, 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 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.

There will be more types of trainers, for instance, multi-agent trainer.

1.2.6 A High-level Explanation

We give a high-level explanation through the pseudocode used in section *Policy*:

```
# pseudocode, cannot work                                    # methods in tianshou
s = env.reset()
buffer = Buffer(size=10000)                                  # buffer = tianshou.
↪data.ReplayBuffer(size=10000)
agent = DQN()                                                # done in policy.__
↪init__(...)
```

(continues on next page)

(continued from previous page)

```
for i in range(int(1e6)):                                    # done in trainer
    a = agent.compute_action(s)                              # done in policy.__
↪call__(batch, ...)
    s_, r, d, _ = env.step(a)                                # done in collector.
↪collect(...)
    buffer.store(s, a, s_, r, d)                             # done in collector.
↪collect(...)
    s = s_                                                   # done in collector.
↪collect(...)
    if i % 1000 == 0:                                        # done in trainer
        b_s, b_a, b_s_, b_r, b_d = buffer.get(size=64)       # done in collector.
↪sample(batch_size)
        # compute 2-step returns. How?
        b_ret = compute_2_step_return(buffer, b_r, b_d, ...) # done in policy.
↪process_fn(batch, buffer, indice)
        # update DQN policy
        agent.update(b_s, b_a, b_s_, b_r, b_d, b_ret)        # done in policy.
↪learn(batch, ...)
```

1.2.7 Conclusion

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

1.3 Tabular Q Learning Implementation

This tutorial shows how to use Tianshou to develop new algorithms.

1.3.1 Background

TODO

1.4 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 :)

1.4.1 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):
```

(continues on next page)

(continued from previous page)

```
    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.

1.4.2 Algorithm specific tricks

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

- *DQNPolicy*: use estimation_step greater than 1 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) of the given batch in each training update will make the algorithm more stable.

1.4.3 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.

1.4.4 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.

1.5 tianshou.data

class tianshou.data.**Batch**(***kwargs*)

Bases: object

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. Here is the usage:

```
>>> import numpy as np
>>> from tianshou.data import Batch
>>> data = Batch(a=4, b=[5, 5], c='2312312')
>>> data.b
[5, 5]
>>> data.b = np.array([3, 4, 5])
>>> len(data.b)
3
>>> data.b[-1]
5
```

In short, you can define a *Batch* with any key-value pair. The current implementation of Tianshou typically use 6 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 return 4 arguments, and the last one is info);

Batch has other methods, including *__getitem__()*, *__len__()*, *append()*, and *split()*:

```
>>> data = Batch(obs=np.array([0, 11, 22]), rew=np.array([6, 6, 6]))
>>> # here we test __getitem__
>>> index = [2, 1]
>>> data[index].obs
array([22, 11])
```

(continues on next page)

(continued from previous page)

```
>>> # here we test __len__
>>> len(data)
3

>>> data.append(data)  # similar to list.append
>>> data.obs
array([0, 11, 22, 0, 11, 22])

>>> # split whole data into multiple small batch
>>> for d in data.split(size=2, permute=False):
...     print(d.obs, d.rew)
[ 0 11] [6 6]
[22  0] [6 6]
[11 22] [6 6]
```

__getitem__(*index*)
Return self[index].

__len__()
Return len(self).

append(*batch*)
Append a *Batch* object to current batch.

split(*size=None*, *permute=True*)
Split whole data into multiple small batch.

Parameters

- **size** (*int*) – if it is None, it does not split the data batch; otherwise it will divide the data batch with the given size. Default to None.
- **permute** (*bool*) – randomly shuffle the entire data batch if it is True, otherwise remain in the same. Default to True.

class tianshou.data.**ReplayBuffer**(*size*)
Bases: object

ReplayBuffer stores data generated from interaction between the policy and environment. It stores basically 6 types of data, as mentioned in *Batch*, based on numpy.ndarray. Here is the usage:

```
>>> 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={})
>>> len(buf)
3
>>> 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.])

>>> 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
```

(continues on next page)

(continued from previous page)

```
# 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])
```

__getitem__(*index*)
Return a data batch: self[index].

__len__()
Return len(self).

add(*obs*, *act*, *rew*, *done*, *obs_next=0*, *info={}*, *weight=None*)
Add a batch of data into replay buffer.

reset()
Clear all the data in replay buffer.

sample(*batch_size*)
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.

update(*buffer*)
Move the data from the given buffer to self.

class tianshou.data.**ListReplayBuffer**
Bases: `tianshou.data.buffer.ReplayBuffer`

The function of *ListReplayBuffer* is almost the same as *ReplayBuffer*. The only difference is that *ListReplayBuffer* is based on `list`.

reset()
Clear all the data in replay buffer.

class tianshou.data.**PrioritizedReplayBuffer**(*size*)
Bases: `tianshou.data.buffer.ReplayBuffer`

docstring for PrioritizedReplayBuffer

add(*obs*, *act*, *rew*, *done*, *obs_next=0*, *info={}*, *weight=None*)
Add a batch of data into replay buffer.

reset()
Clear all the data in replay buffer.

sample(*batch_size*)
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.**Collector**(*policy*, *env*, *buffer=None*, *stat_size=100*, *store_obs_next=True*, ***kwargs*)

Bases: `object`

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

Parameters

- **policy** – an instance of the *BasePolicy* class.
- **env** – an environment or an instance of the *BaseVectorEnv* class.
- **buffer** – an instance of the *ReplayBuffer* class, or a list of *ReplayBuffer*. If set to `None`, it will automatically assign a small-size *ReplayBuffer*.
- **stat_size** (*int*) – for the moving average of recording speed, defaults to 100.
- **store_obs_next** (*bool*) – whether to store the obs_next to replay buffer, defaults to `True`.

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 = VectorEnv([lambda: gym.make('CartPole-v0') for _ in range(3)])
buffers = [ReplayBuffer(size=5000) for _ in range(3)]
# you can also pass a list of replay buffer to collector, for multi-env
# collector = Collector(policy, envs, buffer=buffers)
collector = Collector(policy, envs, buffer=replay_buffer)

# collect at least 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)

# sample data with a given number of batch-size:
batch_data = collector.sample(batch_size=64)
# policy.learn(batch_data)  # btw, vanilla policy gradient only
#   supports on-policy training, so here we pick all data in the buffer
batch_data = collector.sample(batch_size=0)
policy.learn(batch_data)
# on-policy algorithms use the collected data only once, so here we
#   clear the buffer
collector.reset_buffer()
```

For the scenario of collecting data from multiple environments to a single buffer, the cache buffers will turn on automatically. It may return the data more than the given limitation.

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

---

close()
Close the environment(s).

collect(*n_step=0*, *n_episode=0*, *render=0*)
Collect a specified number of step or episode.

Parameters

- **n_step** (*int*) – how many steps you want to collect.
- **n_episode** (*int or list*) – how many episodes you want to collect (in each environment).
- **render** (*float*) – the sleep time between rendering consecutive frames. No rendering if it is 0 (default option).

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()
Return the number of environments the collector has.

render(***kwargs*)
Render all the environment(s).

reset_buffer()
Reset the main data buffer.

reset_env()
Reset all of the environment(s)' states and reset all of the cache buffers (if need).

sample(*batch_size*)
Sample a data batch from the internal replay buffer. It will call *process_fn()* before returning the final batch data.

Parameters **batch_size** (*int*) – 0 means it will extract all the data from the buffer, otherwise it will extract the data with the given batch_size.

seed(*seed=None*)
Reset all the seed(s) of the given environment(s).

1.6 tianshou.env

class tianshou.env.**BaseVectorEnv**(*env_fns*)

Bases: abc.ABC, gym.core.Wrapper

Base class for vectorized environments wrapper. Usage:

```
env_num = 8
envs = VectorEnv([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
```

__len__()

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

abstract close()

Close all of the environments.

abstract render(***kwargs*)

Render all of the environments.

abstract reset(*id=None*)

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

abstract seed(*seed=None*)

Set the seed for all environments. Accept `None`, an int (which will extend `i` to `[i, i + 1, i + 2, ...]`) or a list.

abstract step(*action*)

Run one timestep of all the environments' dynamics. 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 (obs, rew, done, info).

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)

class tianshou.env.**VectorEnv**(*env_fns*)
Bases: tianshou.env.vecenv.BaseVectorEnv

Dummy vectorized environment wrapper, implemented in for-loop. The usage is in *BaseVectorEnv*.

close()
Close all of the environments.

render(***kwargs*)
Render all of the environments.

reset(*id=None*)
Reset the state of all the environments and return initial observations if id is `None`, otherwise reset the specific environments with given id, either an int or a list.

seed(*seed=None*)
Set the seed for all environments. Accept `None`, an int (which will extend `i` to `[i, i + 1, i + 2, ...]`) or a list.

step(*action*)
Run one timestep of all the environments' dynamics. 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 (obs, rew, done, info).

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)

class tianshou.env.**SubprocVectorEnv**(*env_fns*)
Bases: tianshou.env.vecenv.BaseVectorEnv

Vectorized environment wrapper based on subprocess. The usage is in *BaseVectorEnv*.

close()
Close all of the environments.

render(***kwargs*)
Render all of the environments.

reset(*id=None*)
Reset the state of all the environments and return initial observations if id is `None`, otherwise reset the specific environments with given id, either an int or a list.

seed(*seed=None*)
Set the seed for all environments. Accept `None`, an int (which will extend `i` to `[i, i + 1, i + 2, ...]`) or a list.

step(*action*)
Run one timestep of all the environments' dynamics. 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 (obs, rew, done, info).

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)

class tianshou.env.**RayVectorEnv**(*env_fns*)

Bases: tianshou.env.vecenv.BaseVectorEnv

Vectorized environment wrapper based on ray. However, according to our test, it is about two times slower than *SubprocVectorEnv*. The usage is in *BaseVectorEnv*.

close()

Close all of the environments.

render(***kwargs*)

Render all of the environments.

reset(*id=None*)

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

seed(*seed=None*)

Set the seed for all environments. Accept None, an int (which will extend i to [i, i + 1, i + 2, ...]) or a list.

step(*action*)

Run one timestep of all the environments' dynamics. 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 (obs, rew, done, info).

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)

1.7 tianshou.policy

class tianshou.policy.**BasePolicy**(***kwargs*)

Bases: abc.ABC, torch.nn.modules.module.Module

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;
- *`__call__()`*: 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` or `torch.Tensor`), hidden state `state` (for RNN usage), and other information `info` provided by the environment.
2. Output: some `logits` and the next hidden state `state`. 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.

Since *BasePolicy* inherits `torch.nn.Module`, you can operate *BasePolicy* almost the same as `torch.nn.Module`, for instance, load and save the model:

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

abstract __call__(*batch*, *state=None*, ***kwargs*)

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=...)
```

abstract learn(*batch*, ***kwargs*)

Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

process_fn(*batch*, *buffer*, *indice*)

Pre-process the data from the provided replay buffer. Check out *Policy* for more information.

class tianshou.policy.**DQNPolicy**(*model*, *optim*, *discount_factor=0.99*, *estimation_step=1*, *target_update_freq=0*, ***kwargs*)

Bases: tianshou.policy.base.BasePolicy

Implementation of Deep Q Network. arXiv:1312.5602

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).

__call__(*batch*, *state=None*, *model='model'*, *input='obs'*, *eps=None*, ***kwargs*)

Compute action over the given batch data.

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.

More information can be found at *__call__()*.

eval()

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

learn(*batch*, ***kwargs*)

Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

process_fn(*batch*, *buffer*, *indice*)

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

$$G_t = \sum_{i=t}^{t+n-1} \gamma^{i-t}(1-d_i)r_i + \gamma^n(1-d_{t+n}) \max_a Q_{old}(s_{t+n}, \arg\max_a(Q_{new}(s_{t+n}, a)))$$

, where γ is the discount factor, $\gamma \in [0, 1]$, d_t is the done flag of step t. If there is no target network, the Q_{old} is equal to Q_{new}.

set_eps(*eps*)

Set the eps for epsilon-greedy exploration.

sync_weight()

Synchronize the weight for the target network.

train()

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

class tianshou.policy.**PGPolicy**(*model*, *optim*, *dist_fn=<class 'torch.distributions.categorical.Categorical'>*, *discount_factor=0.99*, ***kwargs*)

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** (*torch.distributions.Distribution*) – for computing the action.
- **discount_factor** (*float*) – in [0, 1].

__call__(*batch*, *state=None*, ***kwargs*)
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.

More information can be found at *__call__()*.

learn(*batch*, *batch_size=None*, *repeat=1*, ***kwargs*)
Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

process_fn(*batch*, *buffer*, *indice*)
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, $\gamma \in [0, 1]$.

class tianshou.policy.**A2CPolicy**(*actor*, *critic*, *optim*, *dist_fn=<class 'torch.distributions.categorical.Categorical'>*, *discount_factor=0.99*, *vf_coef=0.5*, *ent_coef=0.01*, *max_grad_norm=None*, ***kwargs*)
Bases: `tianshou.policy.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** (*torch.distributions.Distribution*) – for computing the action, defaults to `torch.distributions.Categorical`.
- **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`.

__call__(*batch*, *state=None*, ***kwargs*)
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.

More information can be found at *__call__()*.

learn(*batch*, *batch_size=None*, *repeat=1*, ***kwargs*)
Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

class tianshou.policy.**DDPGPolicy**(*actor*, *actor_optim*, *critic*, *critic_optim*, *tau=0.005*, *gamma=0.99*, *exploration_noise=0.1*, *action_range=None*, *reward_normalization=False*, *ignore_done=False*, ***kwargs*)
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.
- **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 noise intensity, add to the action, defaults to 0.1.
- **action_range** (*[float, float]*) – the action range (minimum, maximum).
- **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`.

__call__(*batch*, *state=None*, *model='actor'*, *input='obs'*, *eps=None*, ***kwargs*)
Compute action over the given batch data.

Parameters **eps** (*float*) – in [0, 1], for exploration use.

Returns

A *Batch* which has 2 keys:

- `act` the action.
- `state` the hidden state.

More information can be found at *__call__()*.

eval()
Set the module in evaluation mode, except for the target network.

learn(*batch*, ***kwargs*)
Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

process_fn(*batch*, *buffer*, *indice*)
Pre-process the data from the provided replay buffer. Check out *Policy* for more information.

set_eps(*eps*)
Set the eps for exploration.

sync_weight()
Soft-update the weight for the target network.

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

class tianshou.policy.**PPOPolicy**(*actor*, *critic*, *optim*, *dist_fn*, *discount_factor=0.99*, *max_grad_norm=0.5*, *eps_clip=0.2*, *vf_coef=0.5*, *ent_coef=0.0*, *action_range=None*, ***kwargs*)
Bases: tianshou.policy.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** (*torch.distributions.Distribution*) – 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).

__call__(*batch*, *state=None*, *model='actor'*, ***kwargs*)
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.

More information can be found at *__call__()*.

eval()
Set the module in evaluation mode, except for the target network.

learn(*batch*, *batch_size=None*, *repeat=1*, ***kwargs*)
Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

sync_weight()
Synchronize the weight for the target network.

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

class tianshou.policy.**TD3Policy**(*actor*, *actor_optim*, *critic1*, *critic1_optim*, *critic2*, *critic2_optim*, *tau=0.005*, *gamma=0.99*, *exploration_noise=0.1*, *policy_noise=0.2*, *update_actor_freq=2*, *noise_clip=0.5*, *action_range=None*, *reward_normalization=False*, *ignore_done=False*, ***kwargs*)
Bases: tianshou.policy.ddpg.DDPGPolicy

Implementation of Twin Delayed Deep Deterministic Policy Gradient, arXiv:1802.09477

Parameters

- **actor** (*torch.nn.Module*) – the actor network following the rules in *BasePolicy*. (s -> logits)
- **actor_optim** (*torch.optim.Optimizer*) – the optimizer for actor network.
- **critic1** (*torch.nn.Module*) – the first critic network. (s, a -> Q(s, a))
- **critic1_optim** (*torch.optim.Optimizer*) – the optimizer for the first critic network.
- **critic2** (*torch.nn.Module*) – the second critic network. (s, a -> Q(s, a))
- **critic2_optim** (*torch.optim.Optimizer*) – the optimizer for the second critic network.
- **tau** (*float*) – param for soft update of the target network, defaults to 0.005.
- **gamma** (*float*) – discount factor, in [0, 1], defaults to 0.99.
- **exploration_noise** (*float*) – the noise intensity, add to the action, defaults to 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.
- **action_range** (*[float, float]*) – the action range (minimum, maximum).
- **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.

eval()
Set the module in evaluation mode, except for the target network.

learn(*batch*, ***kwargs*)
Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

sync_weight()
Soft-update the weight for the target network.

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

class tianshou.policy.**SACPolicy**(*actor*, *actor_optim*, *critic1*, *critic1_optim*, *critic2*, *critic2_optim*, *tau=0.005*, *gamma=0.99*, *alpha=0.2*, *action_range=None*, *reward_normalization=False*, *ignore_done=False*, ***kwargs*)
Bases: tianshou.policy.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.
- **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 noise intensity, add to the action, defaults to 0.1.
- **alpha** (*float*) – entropy regularization coefficient, default to 0.2.
- **action_range** (*[float, float]*) – the action range (minimum, maximum).
- **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.

__call__(*batch*, *state=None*, *input='obs'*, ***kwargs*)
Compute action over the given batch data.

Parameters **eps** (*float*) – in [0, 1], for exploration use.

Returns

A *Batch* which has 2 keys:

- act the action.
- state the hidden state.

More information can be found at *__call__()*.

eval()
Set the module in evaluation mode, except for the target network.

learn(*batch*, ***kwargs*)
Update policy with a given batch of data.

Returns A dict which includes loss and its corresponding label.

sync_weight()
Soft-update the weight for the target network.

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

1.8 tianshou.trainer

tianshou.trainer.**gather_info**(*start_time*, *train_c*, *test_c*, *best_reward*)
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.**test_episode**(*policy*, *collector*, *test_fn*, *epoch*, *n_episode*)
A simple wrapper of testing policy in collector.

tianshou.trainer.**onpolicy_trainer**(*policy*, *train_collector*, *test_collector*, *max_epoch*, *step_per_epoch*, *collect_per_step*, *repeat_per_collect*, *episode_per_test*, *batch_size*, *train_fn=None*, *test_fn=None*, *stop_fn=None*, *writer=None*, *log_interval=1*, *verbose=True*, *task=''*, ***kwargs*)
A wrapper for on-policy trainer procedure.

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 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 index and performs some operations at the beginning of training in this epoch.
- **test_fn** (*function*) – a function receives the current number of epoch index and performs some operations at the beginning of testing in this epoch.
- **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.

Returns See *gather_info()*.

tianshou.trainer.**offpolicy_trainer**(*policy*, *train_collector*, *test_collector*, *max_epoch*, *step_per_epoch*, *collect_per_step*, *episode_per_test*, *batch_size*, *train_fn=None*, *test_fn=None*, *stop_fn=None*, *writer=None*, *log_interval=1*, *verbose=True*, *task=''*, ***kwargs*)

A wrapper for off-policy trainer procedure.

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 one 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.
- **train_fn** (*function*) – a function receives the current number of epoch index and performs some operations at the beginning of training in this epoch.
- **test_fn** (*function*) – a function receives the current number of epoch index and performs some operations at the beginning of testing in this epoch.
- **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.

Returns See *gather_info()*.

1.9 tianshou.exploration

class tianshou.exploration.**OUNoise**(*sigma=0.3*, *theta=0.15*, *dt=0.01*, *x0=None*)

Bases: object

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*, *mu=0.1*)

Generate new noise. Return a `numpy.ndarray` which size is equal to `size`.

reset()

Reset to the initial state.

1.10 tianshou.utils

class tianshou.utils.**MovAvg**(*size=100*)

Bases: object

Class for moving average. 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*)

Add a scalar into *MovAvg*. You can add `torch.Tensor` with only one element, a python scalar, or a list of python scalar. It will automatically exclude the infinity.

get()

Get the average.

mean()

Get the average. Same as *get()*.

std()
Get the standard deviation.

1.11 Contributing

We always welcome contributions to help make Tianshou better. If you would like to contribute, please check out the guidelines here. Below are an incomplete list of our contributors (find more on this page).

- Jiayi Weng (Trinkle23897)
- Minghao Zhang (Mehooz)

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

G

L

M

O

P

R

S

T

U

V