Tianshou RL library - documentation notes

Marko Guberina 19970821-1256

March 27, 2022

1 Purpose

Great library, has Rainbow which is what i want to use so that I don't have to code it up myself. Also quite neat is the fact that a lot of pedestrian code like logging is already taken care of. The problem is that it's quite involved in the sense that there are many abstractions and layers in the library structure. In other words, there's a lot of code to read to be able to use this thing. As I recently found out, I can't just read complicated technical information and remember it in a retrievable way — hence these notes. Hopefully they'll also be useful as a cheatsheet.

2 Introduction via a Q-learning example

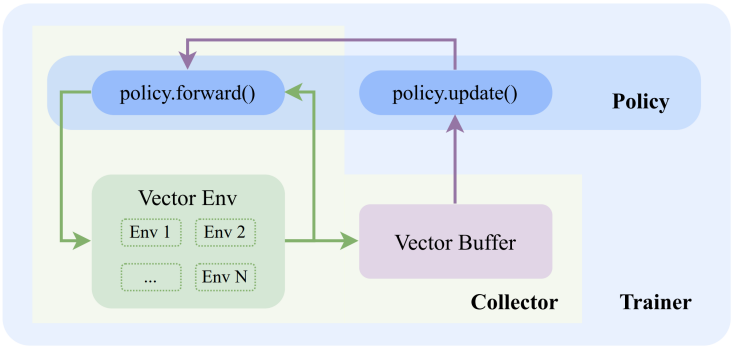

Figure 1: basic q learning pipeline

The gym part is standard. Tianshou supports vectorization for all environments. There are 4 types of vectorizing environment wrappers:

1. `DummyVectorEnv` - sequential version, using a single thread for loop
2. `SubprocVectorEnv` - uses Python multiprocessing and pipes

3. `ShmemVectorEnv` - uses shared memory instead of pipes

4. `RayVectorEnv` - uses Ray. the only choice for computing in a cluster with multiple machines

Policies and **networks** are decoupled — you define a network and pass it to the policy you want. The policies' code is mostly straighforward. **Buffers** are non trivial, as their class implements sampling methods and other related functionality. **Collectors** are a Tianshou specific thing that's very important and not straighforward. The idea behind collectors is to manage how policies interact with the environment — esentially to collect and put data in buffers.

The gist of the collector code is the following:

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

Collectors also enable watching the agent's performance. To do so, you create a collector to which you pass the render argument, for example `render = 1 / 35` for 35 FPS.

There is also a **trainer**, which handles the training, provided everything mentioned so far, including additional training-related parameters. Trainer also supports TensorBoard for logging. You just need to create the logger:

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

and pass it to the trainer.

3 Basic concepts in Tianshou

The control flow of Tianshou's reinforcement learning agent is depicted in the following picture:

3.1 Batch

`Batch` is designed to store and manipulate "hierarchical named tensors". Hierarchical named tensors are a set of tensors whose name forms a hierarchy.

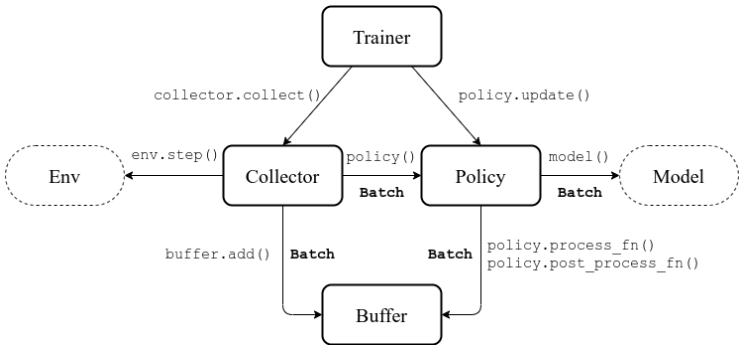

Figure 2: "concepts arch2.png"

Esentially they're nested distionaries. The reason why that's need is because states and actions vary with environments. Consider the usual:

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

`reward` and `done` are mostly just scalar values. But `state` can be a vector, a tensor, a camera input combined with sensory input,... In the last case you need a hierarchical named tensor which could look like:

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

That's all fine and well, but how do you batch these things? Enter `Batch`. It can be defined by using the following rules:

1. a `Batch` can be an empty `Batch()`, or have at leas one key-value pairs (can store keys too, but that's advanced stuff)
2. keys are always strings corresponding to names of corresponding values
3. values can be scalars, tensors, or `Batch` objects. scalars are booleans, ints, floats, the same in NumPy, None and strings

4. both PyTorch `torch.tensor` and NumPy `nd.ndarray` tensors are supported
5. `Batch` can not store dicts! That's internal to `Batch` structure.
6. if you want something edgy, put it in an `np.ndarray` because they support the `np.object` data type and that's pretty much any Python object

There are 7 reserved keys in `Batch`:

1. `obs` — observation at step t
2. `act` — action at step t
3. `rew` — reward at step t
4. `done` — done flag at step t
5. `obs_next` — observation at step $t+1$
6. `info` — info at step t
7. `policy` — data computed by policy at step t

3.1.1 Constructing and using `Batch`es

You can construct them from `dict`ionaries or a list of dictionaries, but the dictionaries in the list will automatically get stacked. You can also do it through `kwargs`, i.e.

```
data = Batch(a=[4, 4], b=[5, 5], c=[None, None])
```

You can also do a combination of the two.

An element can be accessed via `b[key]` or `b.key`. Dictionary methods, like iteration over keys also work if you need those. Furthemore, some methods from NumPy ndarray API are available as well.

Note: to be on the safe side, try out what you're doing in a shell first. You can even stack, concatenate and split them — the values of corresponding keys will be appended in a list of those values under said key.

There are some advanced options which regard reserved keys and heterogenous batches, but I won't be needed those.

The most common use of `Batch` is to be a minibatch from a buffer storing RL experience. Then `Batch` is a collection of tensors whose first dimensions are the same. In that case, `len(b)` is the number of elements in the `Batch` `b`. Likewise you can get the `b.shape`. However if all leaf nodes in a `Batch` are tensors of different lengths, you won't get what you want!

Cool thing: `Batch` is serializable and therefor pickleable.

3.2 Buffer

`ReplayBuffer` stores experience data. It's purpose is to manage `Batch`. All data is stored in a circular queue. Let `buf` be a `ReplayBuffer`. To add to it, do use the `add` member function, ex. `buf.add(Batch(...))`. You can add one buffer to another while keeping chronology with `buf.update(buf2)`. You can also use `load` to load it from a file. For RNN training, you can use frame_stack.

3.3 Policy

There are different classes of policies, but all policies must inherit from `BasePolicy`. Typicall function are the following ones:

1. `__init__()`
2. `forward()` — compute action with given observation
3. `process_fn()` — pre-process data from the replay buffer
4. `learn()` — update policy with a given batch of data
5. `post_process_fn()` — pre-process data from the replay buffer
6. `update()` — this one does it all: samples from buffer, pre-processes data (ex. computing the n-step return), learn from data and post-proces sthe data (ex. update the prioritized replay buffer)

Policies can be in the following states: Because sometimes you want to

Policy state		`policy.training`	`policy.updating`
training state	collecting state	True	False
	updating state	True	True
testing state		False	False

update the (hyper)parameters and sometimes you don't.

`policy.forward` is algorithm specific, but generally it is the mapping of `(batch, state, ...) -> batch`. The input batch comes from either `collect()` or `sample()`.

3.4 Collector

Is quite straighforward, check docs for specific names of things.

3.5 Trainer

Is quite straighforward, check docs for specific names of things. Takes care of ez things like writing for loops for collector, policy updating etc..