Tianshou: a Highly Modularized Deep Reinforcement Learning Library

Jiayi Weng[‡*] TRINKLE23897@GMAIL.COM
Huayu Chen[‡*] CHENHUAY21@MAILS.TSINGHUA.EDU.CN
Dong Yan[‡] SPROBLVEM@GMAIL.COM
Kaichao You[§] YOUKAICHAO@GMAIL.COM
Alexis Duburcq[¶] ALEXIS.DUBURCQ@GMAIL.COM
Minghao Zhang[§] MEHOOZHANG@GMAIL.COM
Hang Su[‡†] SUHANGSS@TSINGHUA.EDU.CN
Jun Zhu[‡†] DCSZJ@TSINGHUA.EDU.CN
[‡] *Dept. of Comp. Sci. & Tech., BNRist Center, Institute for AI, Tsinghua-Bosch Joint ML Center, Tsinghua University, Beijing, 100084 China*
[§] *Department of Software Engineering, Tsinghua University, Beijing, 100084 China*
[¶] *Wandercraft, Paris, France*

Abstract

We present *Tianshou*, a highly modularized python library for deep reinforcement learning (DRL) that uses PyTorch as its backend. Tianshou aims to provide building blocks to replicate common RL experiments and has officially supported more than 15 classic algorithms succinctly. To facilitate related research and prove Tianshou's reliability, we release Tianshou's benchmark of MuJoCo environments, covering 9 classic algorithms and 9/13 Mujoco tasks with state-of-the-art performance. We open-sourced Tianshou at `https://github.com/thu-ml/tianshou/`, which has received over 3k stars and become one of the most popular PyTorch-based DRL libraries.

Keywords: Deep Reinforcement Learning, Modularized, Library, PyTorch, Benchmark

1. Introduction

Recent advances in deep reinforcement learning (DRL) have ignited enthusiasm for this field from both academia and industry. By conjoining deep neural networks and reinforcement learning, DRL has made significant progress in various tasks, ranging from mastering the game of Atari (Mnih et al., 2013) and GO (Silver et al., 2017), to continuous robot controlling in both simulated and real-world scenarios (Duan et al., 2016; Andrychowicz et al., 2020). This progress is accompanied by the flourish of a multitude of newly-emerged DRL algorithms, together with numerous libraries that try to provide reference implementations (Section 2).

However, the architectures of most existing DRL libraries are not well-positioned to meet the requirements of various application scenarios. They often suffer from shortcomings such as lack of flexible and expressive interfaces, complex and redundant code-base, and

∗. J. Weng and H. Chen contributed equally to this article.
†. H. Su and J. Zhu are corresponding authors.

inadequacy of unit tests. To address these challenges, we present Tianshou: a highly modularized python library for deep reinforcement learning based on PyTorch. Tianshou distinguishes itself from existing DRL libraries by highlighting the following characteristics:

Highly modularized Tianshou aims to provide building blocks rather than training scripts. This means that it will focus on designing flexible structures that can be easily customized (Figure 1). In Tianshou, users only need to change a few variables to apply the frequently used techniques in DRL (e.g., parallel data sampling), or to switch to a new algorithm. Also, the separation of Policy (agent) and all other shared infrastructure lightens users' burden of comparing every code-level detail between algorithms.

Reliable While most DRL algorithms are conceptually simple, the state-of-the-art performance of DRL algorithms usually heavily depends on implementation-level details. To prove Tianshou's reliability, we have released a systematic benchmark of Gym's MuJoCo (Brockman et al., 2016; Todorov et al., 2012) 9/13 tasks using 9 model-free agents. By incorporating a comprehensive set of DRL techniques, Tianshou has produced comparable or even better results than the state-of-the-art benchmarks for most algorithms.

Comprehensive Though Tianshou mainly focuses on model-free algorithms, it also supports Multi-agent RL (MARL), Model-based RL (MBRL), and Imitation Learning (IL). Besides that, different data collecting (e.g., both synchronous and asynchronous environment execution) and agent training paradigms in DRL have been formalized through a unified python interface. Lastly, Tianshou has plentiful functionalities that may extend its application, such as compatible with any state or action representation, visualization tools, and full support for recurrent policies. (see Section 3.4).

2. Existing Libraries

Much advance has been made on DRL libraries recently. For example, SpinningUp (Achiam, 2018) gives a simple and straightforward tutorial-style resource as an introduction to DRL. OpenAI Baselines (Dhariwal et al., 2017) and its more elegant version Stable-Baselines (Hill et al., 2018; Raffin et al., 2019) take the lead to provide an efficient RL algorithm implementation. RLlib (Liang et al., 2018) extends the platform to handle distributed training tasks, yet can be complex due to the packaging structure. Dopamine (Castro et al., 2018) mainly focuses on the series of Q-learning approaches and benchmarking Atari Domain (Bellemare et al., 2013). Rlpyt (Stooke and Abbeel, 2019) and ChainerRL (Fujita et al., 2021) are both comprehensive PyTorch-based libraries, aiming to provide modular implementations of classic model-free algorithms. Though the aforementioned libraries are academically successful, most of them could be improved in terms of the flexibility and modularity, code implementation, complete documentation, proper code comments, etc.

3. Architecture of Tianshou

In this section, we will briefly introduce the architecture of Tianshou as illustrated in Figure 1. Tianshou consists of 4 layers, with each layer containing several building blocks. For clarity, we will concentrate on the philosophy behind the design, instead of describing these building blocks one by one.

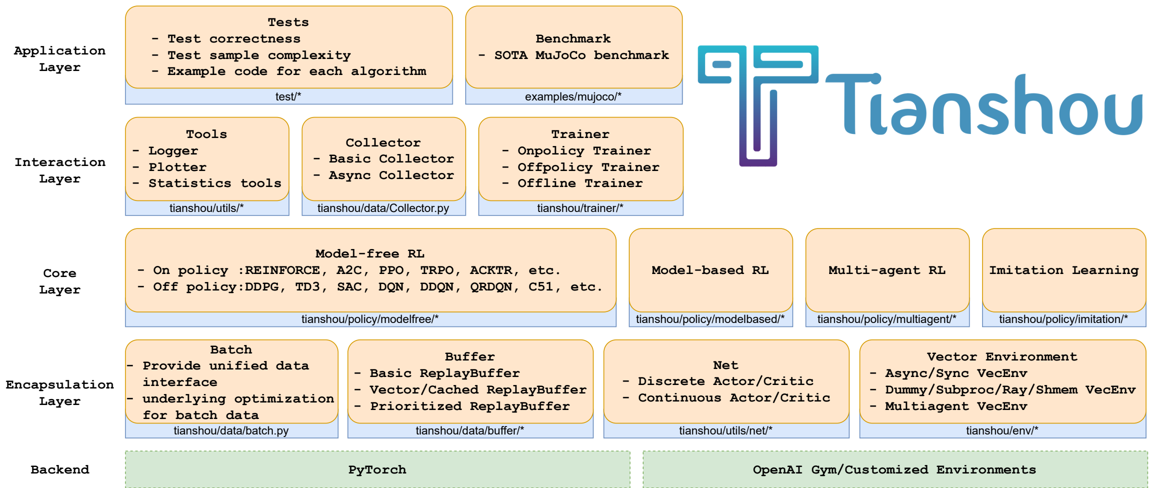

Figure 1: Tianshou's architecture. Using PyTorch and OpenAI/Gym as its backend, Tianshou consists of 4 layers: the bottom-level encapsulation layer provides encapsulations of NumPy arrays, PyTorch modules, Gym environments, etc.; the core layer implements all kinds of DRL algorithms; the interaction layer aims to offer user-oriented high-level APIs, and finally the application layer provides training scripts as examples to demonstrate Tianshou's usage and prove its reliability. We officially implement and support several instances for each building block that share almost the same pythonic APIs (e.g., Buffer is a building block, and Cached ReplayBuffer is one of its instances).

3.1 Standardization of the Training Process

We have standardized and categorized the training paradigms of mainstream DRL algorithms into three types: on-policy training, off-policy training, and offline training, of which each corresponds to an instance of Trainer. We use replay buffer in all three settings to store the transitions, use the collector to collect data into the buffer, and use policy's "update" function to update the agent (Figure 2). Due to the modularization of these underlying building blocks, trainers can be elegantly designed with minimal lines of code through almost the same python APIs. Appendix B presents pseudocode and further details.

3.2 Parallel Computing Infrastructure

Experiment turnaround time is of great importance for research in DRL, which relies heavily on empirical experiments for fast hypothesis validation and hyperparameter searching (Stooke and Abbeel, 2018). However, in contrast to supervised learning that mainly relies on GPU for network optimization, DRL has far more irregularity in computation patterns and typically requires joint computing of CPUs and GPUs (Liang et al., 2018).

In concurrent research, Stooke and Abbeel (2019) address two phases of parallelization in DRL: environment sampling and agent training, while Tianshou focuses on the first one. Our experience is that for medium-scale research, proper parallelization of sampling can usually fulfill researchers' needs, especially when multiple GPUs can be used to run independent experiments with different random seeds (which is often the case in DRL).

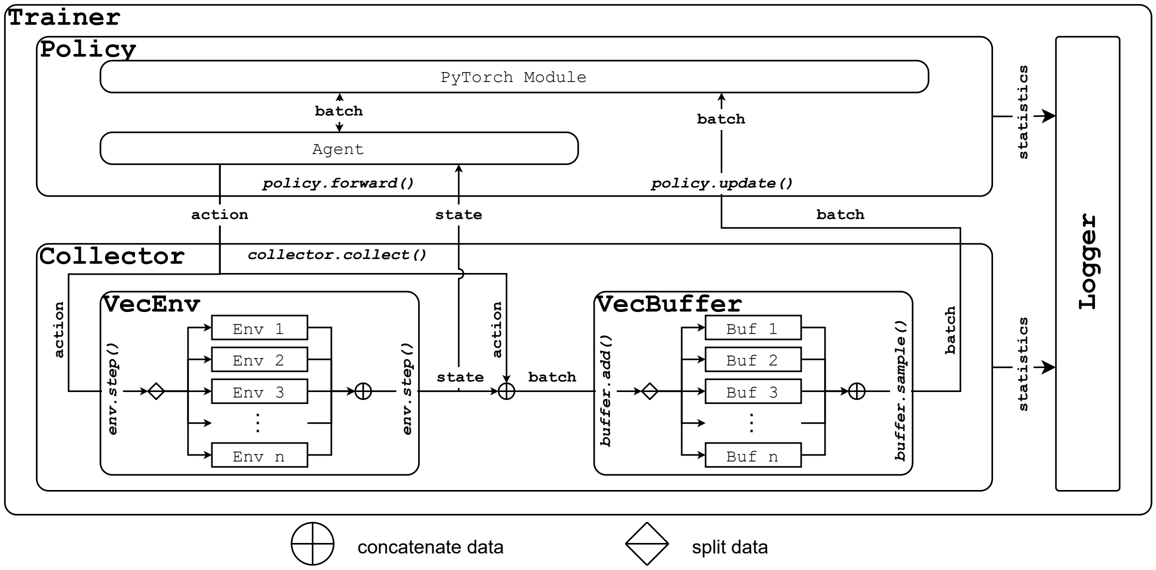

Figure 2: A high-level depiction of Tianshou's training process. Trainer controls all the data flows and interactions between Collector, Policy, and Logger. The collector is in charge of collecting training data, while Policy samples batched data from Vector Buffer and use them to update its agent. Core APIs are highlighted in *italics*. Pseudocode is in Appendix B.

By avoiding communication of network parameters and gradients on different computing nodes, we can keep the code as concise as possible.

Following Clemente et al. (2017), we adopt their parallelization technique to balance simulation and inference loads. Note that our contributions to parallel sampling schemes exceed this work by allowing asynchronized sampling as an alternative way to ease the straggler effect, as opposed to stacking environment instances per process. Figure 2 gives an overview of Tianshou's parallel computing scheme.

Batch A new data structure *Batch* has been introduced to encapsulate different data types (e.g., observation, action, reward, and any other self-defined data). It can be considered a NumPy-compatible dictionary, which supports loop nesting, optimized concatenating and splitting, etc. Batch helps unify interfaces of different algorithms and speed up the calculation of batched data.

VecEnv A series of instances of vectorized environment (*VecEnv*) have been implemented to support parallel data sampling, ranging from dummy VecEnv that is debug-friendly, traditional multi-process VecEnv that can optionally use shared memory for fast communication, to VecEnvs that are specially designed for advanced usage such as multi-agent reinforcement learning.

VecBuffer Replay buffer is a typical data structure widely used in DRL and serves as the medium of interaction between the central training process and worker processes. Like Fujita et al. (2021), Tianshou also supports a handful of advanced usage like stacked or prioritized sampling (Schaul et al., 2015). Traditionally sampled episodes are stored in a circular queue, but we go a bit further by not restricting how data is stored. This is for the reason that different algorithms might have different data access requirements.

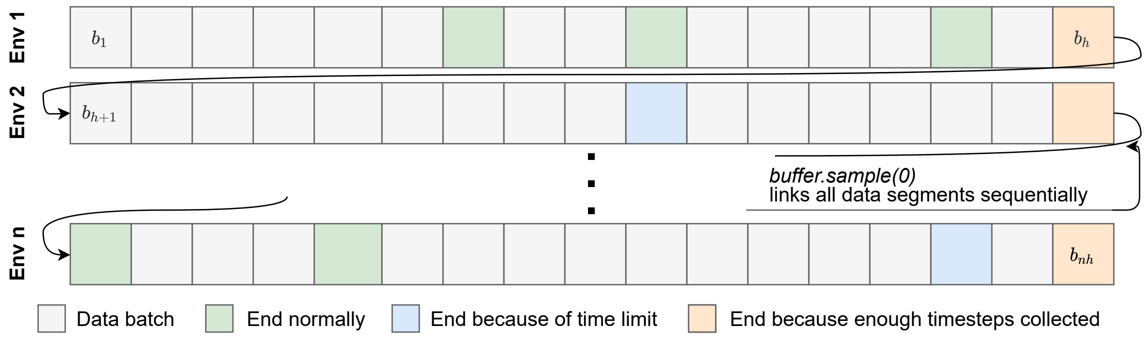

Figure 3: An illustration of how batched data is stored in and sampled from VectorReplay-Buffer (an instance of ReplayBuffer). The trajectories collected from one specific environment are stored in a fixed and independent circular queue. When calling *buffer.sample(0)*, all trajectories will be linked sequentially, even if some trajectories are still incomplete.

For instance, on-policy algorithms would prefer data steps from the same episode grouped in sequential order for convenience when computing reward-to-go. In contrast, off-policy algorithms that look one step ahead might not have such a need. Besides, it's natural to store interaction data sampled from one sub-process worker in an individual sub-buffer, yielding Vector Replay Buffer (*VecBuffer*) and Cached Replay Buffer.

All instances of Replay Buffers share the same APIs, which can be used to compute values like GAE advantages (Schulman et al., 2015) and n-step returns (De Asis et al., 2018) without knowing the explicit underlying storage mechanism. The goal is to maintain a well-encapsulated architecture so that users don't have to care whether the underlying parallel optimization is used when implementing their own algorithm.

3.3 Detail Handling

Tianshou aims to relieve users from imperceptible details critical to good performance and, hence, incorporates a comprehensive set of DRL techniques. Techniques include proper handling of truncated episodes by bootstrapping (Pardo et al., 2018), value normalization (van Hasselt et al., 2016), automatic action scaling, and clipping, as its infrastructure. Since these techniques are relatively scattered, we will take multi-segment bootstrapping as an example to shed some light on how Tianshou deals with these code-level details.

Generalized advantage (Schulman et al., 2015) is a fundamental quantity frequently used when estimating the gradient of the neural network's parameter θ, and can be viewed as a discounted sum of Bellman residual terms:

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$$

$$\delta_t^V = r_t + \gamma V\left(s_{t+1}\right) - V\left(s_t\right)$$

where V is a value estimator, and both γ and λ are discount factors. State s_t and reward r_t, together with action a_t and done flag d_t are quantities at time-step t in the collected trajectory $\{s_1, a_1, r_1, d_1, s_2, a_2, r_2, d_2, s_3 ...\}$.

Timestep Type	Ordinary	Last(N)	Last(T)	Last(E)
Done flag d_t	*False*	*True*	*True*	*False*
Index in *buffer.unfinished_index()*?	*False*	*True*	*True*	*Ture*
Need bootstrapping?	*False*	*False*	*True*	*True*
BasePolicy.value_mask() of the Index	*False*	*False*	*True*	*True*

Table 1: Different types of timestep batches and their characteristics. From left to right, four timestep types correspond to the gray, green, blue, and orange blocks in Figure 3.

The above formula assumes s_{t+1} always exists. A common approach to calculating δ_t^V when t is already the last step of an episode is to simply regard δ_{t+1}^V as 0. However, as indicated in Figure 3, there are multiple possibilities when an episode ends. Pardo et al. (2018) has shown that giving up bootstrapping whenever the done flag is met will lead to sub-optimal policies and training instability if the episode is truncated because of the time limit. They proposed the *partial-episode bootstrapping* strategy, which Tianshou adopts and extends so that it also works when episodes are temporarily truncated before updating (see Table 1 and Algorithm 1 for details). The same methods also apply to calculate n-steps returns and discounted reward-to-go, and can be implemented elegantly using shared APIs provided by *Buffer*.

Algorithm 1: Generalized Advantage Estimation for Partial-Episodes

for $t \leftarrow nh$ **downto** 1 **do**

$\delta_t^V = r_t + \mathit{BasePolicy.value\_mask}(b_t) \times \gamma V(s_{t+1}) - V(s_t)$;

$\hat{A}_t = \delta_t^V$ **if** t in *buffer.unfinished_index()* **else** $\delta_t^V + \gamma\lambda\hat{A}_{t+1}$;

end

3.4 Utilities

Besides the building blocks menti1oned above, Tianshou has plentiful extra features or functionalities that users might find helpful. For instance, Tianshou has customizable loggers that are tensorboard-compatible, plotting and data analysis tools to assist ablation studies, and pre-defined network templates. N-step returns and GAE, as illustrated above, and automatic action scaling are all handy built-in methods inherited from a base class. Recurrent state representation, prioritized experience replay, training resumption, and buffer data import/export are also supported.

4. Performance

Reproducibility is a long-standing problem in DRL. Numerous low-level design choices that are not extensively discussed make it hard to attribute progress in DRL. This is because sometimes imperceptible details, instead of what are emphasized in papers might be the key factor for good performance (Pardo et al., 2018; Engstrom et al., 2019).

To facilitate related research and prove Tianshou's reliability, we have released Tianshou's benchmark of OpenAI Gym MuJoCo task suite (Table 2), featuring 9 classic on-and-off policy algorithms supported by Tianshou in 9 out of 13 MuJoCo tasks. We selected these

Task	Ant-v3	HalfCheetah-v3	Hopper-v3	Walker2d-v3	Humanoid-v2
DDPG (1M)	990.4 ± 4.3	11718.7 ± 465.6	2197.0 ± 971.6	1400.6 ± 905.0	177.3 ± 77.6
TD3 (1M)	5116.4 ± 799.9	10201.2 ± 772.8	3472.2 ± 116.8	3982.4 ± 274.5	5189.5 ± 178.5
SAC (1M)	5850.2 ± 475.7	12138.8 ± 1049.3	3542.2 ± 51.5	5007.0 ± 251.5	5488.5 ± 81.2
REINFORCE (10M)	1108.1 ± 323.1	1138.8 ± 104.7	416.0 ± 104.7	440.9 ± 148.2	464.3 ± 58.4
A2C (3M)	5236.82 ± 36.7	2377.3 ± 1363.7	1608.6 ± 529.5	1805.4 ± 1055.9	5316.6 ± 554.8
PPO (3M)	4079.3 ± 880.2	7337.4 ± 1508.2	3127.7 ± 413.0	4895.6 ± 704.3	1359.7 ± 572.7
NPG (3M)	4820.6 ± 318.6	5198.7 ± 430.0	2611.1 ± 554.5	4459.5 ± 803.6	4391.9 ± 1245.5
TRPO (3M)	4854.4 ± 421.3	5792.1 ± 656.0	2626.67 ± 778.9	4787.1 ± 609.2	5154.4 ± 594.1
ACKTR (3M)	3945.8 ± 965.3	2520.1 ± 1368.7	1924.4 ± 704.9	1329.1 ± 498.9	361.3 ± 38.5

Table 2: Tianshou's MuJoCo benchmark. Max average return over 10 trails (different random seeds) of collected timesteps indicated in the brackets ± a single standard deviation over trails is given. For each trial, performance is averaged on another 10 test seeds. We evaluate algorithms in a stochastic way (actions are sampled from distributions) at regular intervals. Further details appear in Appendix A.

tasks because they are the most commonly used ones in the literature. Compared to the already heavily benchmarked Atari domain, it is typically lacking a published and detailed benchmark on the MuJoCo task suite, which is also frequently used in research. As in Figure 5 in Appendix A, by comparison with both classic literature and open source implementations, Tianshou's performances are roughly at-parity with or better than the best-reported results for most of the algorithms in MuJoCo settings and are suitable to be used for research purposes. For each algorithm benchmark, we provide raw training data, log details, a pre-trained agent, and some hints on why we choose certain key hyperparameters or how to tune the algorithm. Detailed training curves can be found in Appendix A. A more detailed version is available at `https://github.com/thu-ml/tianshou/tree/master/examples/mujoco`.

Some libraries (Fujita et al., 2021) devote to faithfully replicating existing papers, while Tianshou aims to present an as-consistent-as-possible set of hyperparameters and low-level designs. While leaving the core algorithm untouched, we try to incorporate several tricks that are known helpful in a specific algorithm to all similar algorithms supported by Tianshou. Hopefully, this will help build a baseline that is not too easy to be crossed and make comparisons between algorithms more objective.

5. Usability

Tianshou is lightweight and easy to install. It only depends on *NumPy*, *Gym*, *Tqdm*, *numba*, *h5py* and *PyTorch*. Users can simply install Tianshou with command `pip install tianshou` or `conda install tianshou -c conda-forge`.

Tinashou is friendly to beginners. Full API documentation and a series of tutorials are provided at `https://tianshou.readthedocs.io/`. Only a few lines of code are required to start a simple experiment. Besides, Tianshou is elegant. It strictly follows the PEP8 python code style and attaches great importance to code comments and type annotations. Tianshou has extensive unit and integration tests with GitHub Actions, including PEP8 code-style check, type check, and performance tests on different platforms (Windows, MacOS, Linux). The code-coverage report is available at `https://codecov.io/gh/thu-ml/tianshou`.

6. Conclusion

This manuscript briefly describes Tianshou, a flexible, reliable and stable implementation of a modular DRL library. The philosophy behind the design of Tianshou is explained, together with its functionality and usage. Tianshou tries to set up a framework for DRL research by providing a series of elegantly designed building blocks and their instances. We have released a detailed benchmark for OpenAI MuJoCo environments, covering most of their tasks and many classic algorithms, demonstrating Tianshou's reliability.

Acknowledgments

We thank Haosheng Zou for his pioneering work of tensorflow-based Tianshou before version 0.1.1. We thank Peng Zhong, Qiang He, Yi Su, Qing Xiao, Qifan Li, Yan Li, and all other people for their great contributions to Tianshou.

References

Joshua Achiam. Spinning Up in Deep Reinforcement Learning. `https://github.com/openai/spinningup`, 2018.

OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G. Bellemare. Dopamine: A research framework for deep reinforcement learning. *CoRR*, abs/1812.06110, 2018.

Alfredo V Clemente, Humberto N Castejón, and Arjun Chandra. Efficient parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1705.04862*, 2017.

Kristopher De Asis, J Hernandez-Garcia, G Holland, and Richard Sutton. Multi-step reinforcement learning: A unifying algorithm. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International conference on machine learning*, pages 1329–1338. PMLR, 2016.

Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation matters in deep rl: A case study on ppo and trpo. In *International conference on learning representations*, 2019.

Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596. PMLR, 2018.

Yasuhiro Fujita, Prabhat Nagarajan, Toshiki Kataoka, and Takahiro Ishikawa. Chainerrl: A deep reinforcement learning library. *Journal of Machine Learning Research*, 22(77): 1–14, 2021.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870. PMLR, 2018.

Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael I. Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 3059–3068, 2018.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Fabio Pardo, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. Time limits in reinforcement learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4045–4054. PMLR, 10–15 Jul 2018. URL `http://proceedings.mlr.press/v80/pardo18a.html`.

Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable baselines3. `https://github.com/DLR-RM/stable-baselines3`, 2019.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

Adam Stooke and Pieter Abbeel. Accelerated methods for deep reinforcement learning. *arXiv preprint arXiv:1803.02811*, 2018.

Adam Stooke and Pieter Abbeel. rlpyt: A research code base for deep reinforcement learning in pytorch. *arXiv preprint arXiv:1909.01500*, 2019.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

Hado van Hasselt, Arthur Guez, Matteo Hessel, Volodymyr Mnih, and David Silver. Learning values across many orders of magnitude. *arXiv preprint arXiv:1602.07714*, 2016.

Yuhuai Wu, Elman Mansimov, Shun Liao, Roger Grosse, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. *arXiv preprint arXiv:1708.05144*, 2017.

Appendix A. MuJoCo Benchmark of Tianshou

In this appendix, we present further details of Tianshou's Mujoco Benchmark, including a brief runtime analysis, plot figures, quantitative results on 4 tasks not mentioned in Table 2, and comparison with other published benchmarks. All results are obtained using a single Nvidia TITAN X GPU and up to 48 CPU cores.

Algorithm	# of Parallel Envs	Total (1M timesteps)	Collecting (%)	Updating (%)	Evaluating (%)	Others (%)
DDPG	1	2.9h	12.0	80.2	2.4	5.4
TD3	1	3.3h	11.4	81.7	1.7	5.2
SAC	1	5.2h	10.9	83.8	1.8	3.5
REINFORCE	64	4min	84.9	1.8	12.5	0.8
A2C	16	7min	62.5	28.0	6.6	2.9
PPO	64	24min	11.4	85.3	3.2	0.2
NPG	16	7min	65.1	24.9	9.5	0.6
TRPO	16	7min	62.9	26.5	10.1	0.6
ACKTR	16	11min	45.2	46.6	6.1	2.0

Table 3: Runtime averaged on 9 benchmarked Mujoco tasks.

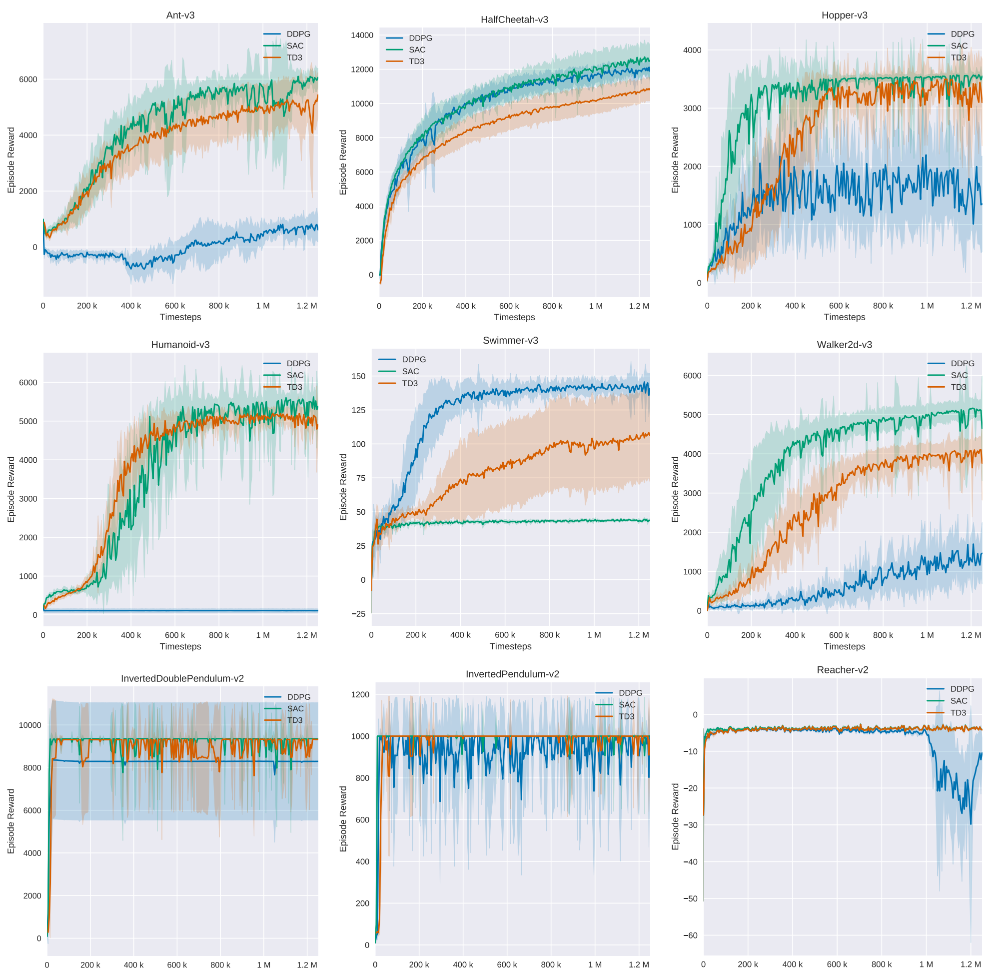

Figure 4: Learning curves of Tianshou's off-policy algorithms for the OpenAI gym MuJoCo tasks. Average returns over 10 trails ± a single standard deviation over trails are illustrated.

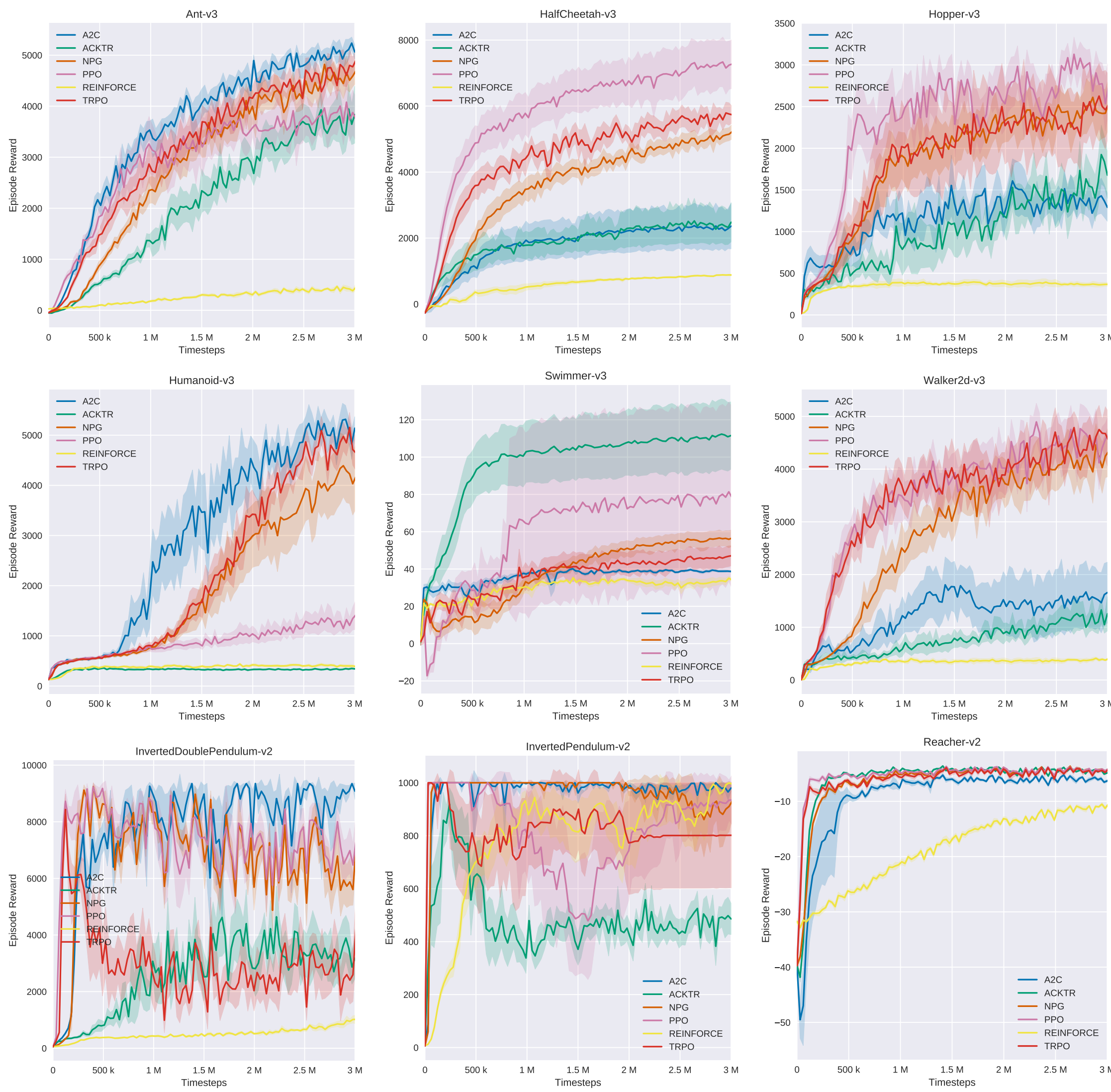

Figure 5: Learning curves of Tianshou's on-policy algorithms for the OpenAI gym MuJoCo tasks. Average returns over 10 trails ± half a single standard deviation over trails are illustrated.

Task	Swimmer-v3	Reacher-v3	InvertedPendulum-v2	InvertedDoublePendulum-v2
DDPG (1M)	144.1 ± 6.5	-3.3 ± 0.3	1000.0 ± 0.0	8364.3 ± 2778.9
TD3 (1M)	104.2 ± 34.2	-2.7 ± 0.2	1000.0 ± 0.0	9349.2 ± 14.3
SAC (1M)	44.4 ± 0.5	-2.6 ± 0.2	1000.0 ± 0.0	9359.5 ± 0.4
REINFORCE (10M)	35.6 ± 2.6	-5.5 ± 0.2	1000.0 ± 0.0	7726.2 ± 1287.3
A2C (3M)	40.2 ± 1.8	-5.2 ± 0.5	1000.0 ± 0.0	9351.3 ± 12.8
PPO (3M)	81.4 ± 96.0	-3.7 ± 0.3	1000.0 ± 0.0	9231.3 ± 270.4
NPG (3M)	56.6 ± 8.7	-3.6 ± 0.3	1000.0 ± 0.0	9243.2 ± 276.0
TRPO (3M)	47.0 ± 10.8	-3.9 ± 0.4	1000.0 ± 0.0	8435.2 ± 1073.3
ACKTR (3M)	112.3 ± 37.7	-3.6 ± 0.3	919.4 ± 122.2	4644.2 ± 2165.4

Table 4: Tianshou's MuJoCo Benchmark. This serves as a supplement to Table 2.

Task		Ant	HalfCheetah	Hopper	Walker2d	Swimmer	Humanoid	Reacher	IPendulum	IDPendulum
DDPG	**Tianshou**	990.4	**11718.7**	**2197.0**	1400.6	**144.1**	**177.3**	**-3.3**	**1000.0**	8364.3
	Achiam (2018)	∼840	∼11000	∼1800	∼1950	∼137	-	-	-	-
	Fujimoto et al. (2018)	**1005.3**	3305.6	2020.5	1843.6	-	-	-6.5	**1000.0**	**9355.5**
	Fujimoto et al. (2018)(Our)	888.8	8577.3	1860.0	**3098.1**	-	-	-4.0	**1000.0**	8370.0
TD3	**Tianshou**	**5116.4**	**10201.2**	3472.2	3982.4	**104.2**	**5189.5**	**-2.7**	**1000.0**	**9349.2**
	Achiam (2018)	∼3800	∼9750	∼2860	∼4000	∼78	-	-	-	-
	Fujimoto et al. (2018)	4372.4	9637.0	**3564.1**	**4682.8**	-	-	-3.6	**1000.0**	9337.5
SAC	**Tianshou**	**5850.2**	**12138.8**	**3542.2**	**5007.0**	**44.4**	**5488.5**	**-2.6**	**1000.0**	**9359.5**
	Achiam (2018)	∼3980	∼11520	∼3150	∼4250	∼41.7	-	-	-	-
	Haarnoja et al. (2018)	∼3720	∼10400	∼3370	∼3740	-	∼5200	-	-	-
	Fujimoto et al. (2018)	655.4	2347.2	2996.7	1283.7	-	-	-4.4	**1000.0**	8487.2
A2C	**Tianshou**	**3485.4**	**1829.9**	**1253.2**	**1091.6**	**36.6**	**1726.0**	**-6.7**	**1000.0**	**9257.7**
	Schulman et al. (2017)	-	∼1000	∼900	∼850	∼31	-	∼-24	**∼1000**	∼7100
	Schulman et al. (2017)(TR)	-	∼930	∼1220	∼700	**∼36**	-	∼-27	**∼1000**	∼8100
PPO	**Tianshou**	**3258.4**	**5783.9**	**2609.3**	**3588.5**	66.7	**787.1**	**-4.1**	**1000.0**	**9231.3**
	Schulman et al. (2017)	-	∼1800	∼2330	∼3460	∼108	-	∼-7	**∼1000**	∼8000
	Fujimoto et al. (2018)	1083.2	1795.4	2164.7	3317.7	-	-	-6.2	**1000.0**	8977.9
	Dhariwal et al. (2017)	-	∼1700	∼2400	∼3510	∼111	-	∼-6	∼940	∼7350
	Achiam (2018)	∼650	∼1670	∼1850	∼1230	**∼120**	-	-	-	-
TRPO	**Tianshou**	**2866.7**	**4471.2**	2046.0	**3826.7**	40.9	**810.1**	**-5.1**	**1000.0**	**8435.2**
	Wu et al. (2017)	∼0	∼400	∼1400	∼550	∼40	-	-8	**∼1000**	∼800
	Fujimoto et al. (2018)	-75.9	-15.6	**2471.3**	2321.5	-	-	-111.4	985.4	205.9
	Schulman et al. (2017)	-	∼0	∼2100	∼1100	**∼121**	-	∼-115	**∼1000**	∼200
	Dhariwal et al. (2017)	-	∼1350	∼2200	∼2350	∼95	-	**∼-5**	∼910	∼7000
	Achiam (2018)(Tensorflow)	∼150	∼850	∼1200	∼600	∼85	-	-	-	-
ACKTR	**Tianshou**	1368.2	1876.3	1066.2	625.0	**101.6**	**361.3**	**-3.8**	919.4	3073.2
	Wu et al. (2017)	∼1350	**∼2450**	**∼3500**	∼620	∼42	-	∼5	**∼1000**	**∼9300**
	Fujimoto et al. (2018)	**1821.9**	1450.5	2428.4	**1216.7**	-	-	-4.3	**1000.0**	9081.9

Table 5: The performance of Tianshou against published results on OpenAI Gym MuJoCo benchmarks. We use max average return in 1M timesteps as the reward metric. ∼ means the result is approximated from the plots because quantitative results are not provided. - means results are not provided. The best-performing baseline on each task is highlighted in boldface. Please note that the versions of MuJoCo tasks and the number of trails used may vary across different baselines. In Tianshou, we always use the latest task versions in OpenAI Gym and 10 trails for evaluation, while other baselines may use fewer trails.

Appendix B. Pseudocode

In this appendix, we provide pseudocode of an instance for each of the three building blocks in Figure 2. The workflow of an off-policy trainer is depicted in Algorithm 2. For the on-policy trainer, the main difference is that we clear the buffer after Line 10. The collector can collect either a specified number of timesteps or episodes (Algorithm 3). When updating Policy (Algorithm 4), *Policy.process_fn* is usually used to calculate quantities like GAE advantage and n-step return. Typical usage of *Policy.post_process_fn* is to update the sampling weight in the prioritized buffer. Note that users usually don't have to implement these two methods themselves and can simply inherit them from *BasePolicy* provided by Tianshou.

Algorithm 2: Off-policy Trainer

Input: n-epochs N, steps-per-epoch L, steps-per-collect l, updates-per-step u
Data: policy, collector, test_collector, logger.

1: **repeat** N **times**
2:     T = 0;
3:     **while** $T < L$ **do**
4:         Collect l steps by running *result = collector.collect(n_step(episode) = l)*, training data goes to *collector.buffer* and statistics to *result*;
5:         *T += result['n_steps']* because we might not exactly collect l steps;
6:         *logger.log_train_data(result)* at regular interval;
7:         **repeat** *round(u∗result['n_steps'])* **times**
8:             Update policy: *losses = policy.update(collector.buffer)*;
9:             *logger.log_update_data(losses)* at regular interval;
10:         **end**
11:     **end**
12:     Evaluate: *logger.log_test_data(test_collector.collect(n_episode= #test_seeds))*;
13: **end**

Algorithm 3: Synchronous Collector

Input: n-episode(step) l, random-action (default to False)
Data: policy, envs, buffer

if *envs hasn't been reset yet* **then** *self.reset()*, initial states go to a batch: *self.data*;
Set up T = 0(E = 0) to track the number of already collected timesteps(epsisodes);
while $T < l$ **do**
    action = stack(envs.action_space.sample()) **if** *random* **else** *policy(self.data);*
    Add exploration noise to action if *policy.exploration_noise()* is defined;
    action ⇢ *self.data*;
    Map action (DNN space to env space) by calling *policy.map_action(self.data)*;
    envs.step(remapped_action) ⇢ *self.data*;
    buffer.add(self.data) and collect statistics returned by the buffer;
    Reset finished envs;
    Clear part of *self.data*;
end
Return statistics (e.g. rewards and start indices of each episode) of this collection;

Algorithm 4: Policy.update

Input: buffer, sample-size (0 for on-policy algorithms)
Data: policy

Sample some data for updating: *batch, indice = buffer.sample(sample_size)*;
with *policy* **in training mode**
  Prepare *batch* to be ready for updating: *self.process_fn(batch, buffer, indice)*;
  Update network parameters θ: *self.learn(batch)*;
  Do remaining work: *self.post_process_fn(batch, buffer, indice)*;
end

Appendix C. Usage: A Case Study

In this appendix, we use an example to show how Tianshou can be used to run a common DRL experiment with minimal lines of code.

```python
# Original implementation can be found at examples/mujoco/mujoco_reinforce.py
import gym
import torch
import numpy as np

from tianshou.policy import PGPolicy
from tianshou.utils import BasicLogger
from tianshou.utils.net.common import Net
from tianshou.trainer import onpolicy_trainer
from tianshou.utils.net.continuous import ActorProb
from tianshou.data import Collector, VectorReplayBuffer
from tianshou.env import ShmemVectorEnv, SubprocVectorEnv

# Training environments that has 16 asynchronous parallel workers, with shared
# memory for fast communication, and timeout=0.1s to reduce straggler effect.
# Testing environments uses 10 synchronous parallel workers. Both have automatic
# observation normalization.
envs = ShmemVectorEnv(
    [lambda: gym.make("task_name") for _ in range(16)],
    norm_obs=True, timeout=0.1)
test_envs = SubprocVectorEnv(...)

# Define a MLP mod2el (#state * 64 * 64 * #action) with tanh activation
state_shape, ... = envs.observation_space.shape, ...
preprocess_net = Net(state_shape, hidden_sizes=(64, 64), activation="Tanh")
actor = ActorProb(preprocess_net, ...)

# Optimizer that uses linear learning rate scheduler
optim = torch.optim.Adam(actor.parameters(), ...)
lr_scheduler = torch.optim.lr_scheduler.LambdaLR(optim, ...)

# Define Gaussion distribution, from which action is sampled.
from torch.distributions import Independent, Normal
```

```
def dist(*logits):
    return Independent(Normal(*logits), 1)

# Traditional REINFORCE agent that use value normalization, with
# automatic action scaling/squashing and learning rate decay.
policy = PGPolicy(actor, optim, dist, discount_factor=0.99,
            reward_normalization=True, action_scaling=True,
            action_bound_method="tanh", lr_scheduler=lr_scheduler)

# load a pre-trained policy (if provided)
if resume_path:
    policy.load_state_dict(torch.load(resume_path))

# Set up collector for training (with a 10*1e5 vector buffer) and testing.
buffer = VectorReplayBuffer(buffer_size=1e6, buffer_num=len(envs))
train_collector = Collector(policy, envs, buffer, ...)
test_collector = Collector(policy, test_envs)

# Set up tensorboard-based logger to record training details to log_path.
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter(log_path)
logger = BasicLogger(writer, update_interval=10, train_interval=100)

# The trainer below trains the agent for 100 epochs. For each epoch, it
# collects 1000 steps from 16 parallel workers and then update the agent
# until 4000 steps are collected.
result = onpolicy_trainer(policy, train_collector, test_collector,
            epoch=100, step_per_epoch=4000, step_per_collect=1000, ...)
print(result)
```

Appendix D. Comparison

Libraries	*Baselines*	*ChainerRL*	*ray/rllib*	*Stable Baselines 3*	*rlpyt*	*Tianshou*
Backend	Tensorflow	Chainer	Tensorflow/Pytorch	Pytorch	Pytorch	Pytorch
Github Stars	11.7k	1.0k	16.7k (ray)	1.9k	1.9k	3.3k
# of Alg.	9	13	>14	7	>3	>20
RNN Support		✓	✓		✓	✓
Visualization Tools	✓	✓	✓	✓	✓	✓
PEP8 code style		✓	✓	✓	✓	✓
Type Hints				✓		✓
Code Coverage	-	81%	-	96%	15%	95%
Documentation		✓	✓	✓	✓	✓
Last Update	2020/01	2021/04	2021/07	2021/07	2020/09	2021/07

Table 6: A comparison between Tianshou and other popular DRL libraries (by 2021/07).