Journal of Machine Learning Research 23 (2022) 1-18 Submitted 11/21; Revised 6/22; Published 7/22

CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms

Shengyi Huang[1] COSTA.HUANG@OUTLOOK.COM
Rousslan Fernand Julien Dossa[2] DOSS@AI.CS.KOBE-U.AC.JP
Chang Ye[3] C.YE@NYU.EDU
Jeff Braga[1] JEFFREYBRAGA@GMAIL.COM
Dipam Chakraborty[4] DIPAMC77@GMAIL.COM
Kinal Mehta[5] KINAL.MEHTA11@GMAIL.COM
João G.M. Araújo[6] JOAOGUILHERMEARUJO@GMAIL.COM
[1] *College of Computing and Informatics, Drexel University, USA*
[2] *Graduate School of System Informatics, Kobe University, Japan*
[3] *Tandon School of Engineering, New York University, USA*
[4] *AIcrowd*
[5] *International Institute of Information Technology, Hyderabad*
[6] *Cohere*

Editor: Joaquin Vanschoren

Abstract

`CleanRL` is an open-source library that provides high-quality single-file implementations of Deep Reinforcement Learning (DRL) algorithms. These single-file implementations are self-contained algorithm variant files such as `dqn.py`, `ppo.py`, and `ppo_atari.py` that individually include all algorithm variant's implementation details. Such a paradigm significantly reduces the complexity and the lines of code (LOC) in each implemented variant, which makes them quicker and easier to understand. This paradigm gives the researchers the most fine-grained control over all aspects of the algorithm in a single file, allowing them to prototype novel features quickly. Despite having succinct implementations, CleanRL's codebase is thoroughly documented and benchmarked to ensure performance is on par with reputable sources. As a result, CleanRL produces a repository tailor-fit for two purposes: 1) understanding all implementation details of DRL algorithms and 2) quickly prototyping novel features. CleanRL's source code can be found at `https://github.com/vwxyzjn/cleanrl`.

Keywords: deep reinforcement learning, single-file implementation, open-source

1. Introduction

In recent years, Deep Reinforcement Learning (DRL) algorithms have achieved great success in training autonomous agents for tasks ranging from playing video games directly from pixels to robotic control (Mnih et al., 2013; Lillicrap et al., 2016; Schulman et al., 2017). At the same time, open-source DRL libraries also flourish in the community (Raffin et al., 2021; Liang et al., 2018; D'Eramo et al., 2020; Fujita et al., 2021; Weng et al., 2021). Many of them have adopted good modular designs and fostered vibrant development communities. Nevertheless, understanding all the implementation details of an algorithm remains difficult

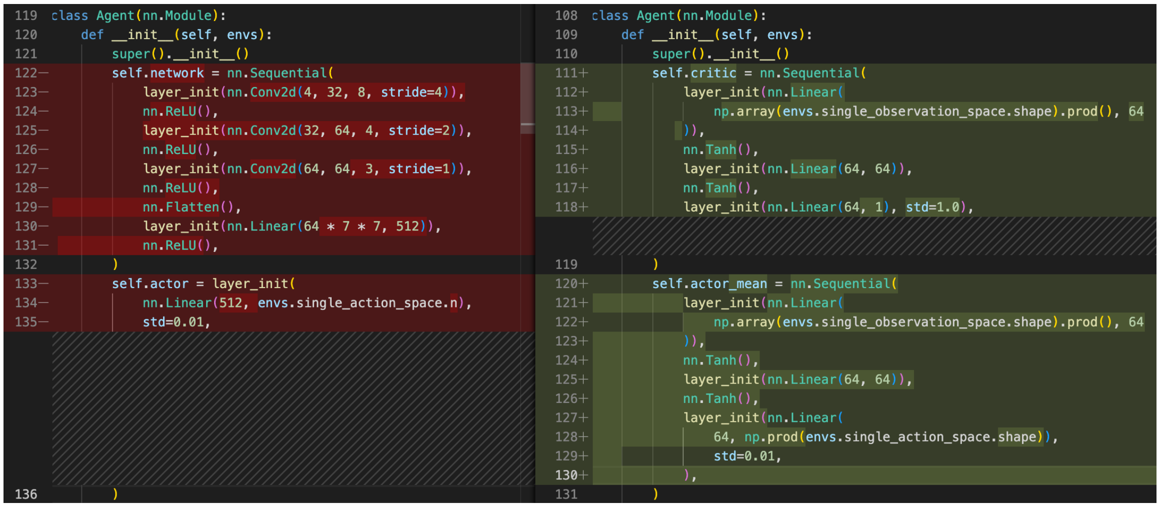


Figure 1: Filediff in Visual Studio Code: left click select `ppo_atari.py` then `cmd/ctrl` + left click select `ppo_continuous_action.py` to highlight neural network architecture differences of PPO when applying to Atari games and MuJoCo tasks.

because these details are spread to different modules. However, understanding these implementation details is essential because they could significantly affect performance (Engstrom et al., 2020).

In this paper, we introduce `CleanRL`, a DRL library based on single-file implementations to help researchers understand all the details of an algorithm, prototype new features, analyze experiments, and scale the experiments with ease. `CleanRL` is a *non-modular* library. Each algorithm variant in `CleanRL` is self-contained in a single file, in which the lines of code (LOC) have been trimmed to the bare minimum. Along with succinct implementations, CleanRL's codebase is thoroughly documented and benchmarked to ensure performance is on par with reputable sources. For example, our Proximal Policy Optimization (PPO) (Schulman et al., 2017) implementation with Atari games is a single file `ppo_atari.py` using only 337 LOC, yet it closely matches `openai/baselines`' PPO performance in the game breakout (Appendix A), making it much easier to understand the algorithm in one go. In contrast, the researchers using modular DRL libraries often need to understand the modular design (usually 7 to 20 files) which can contain thousands of LOC. As a result, CleanRL is tailor-fit for two purposes: 1) understanding all implementation details of DRL algorithms and 2) quickly prototyping novel features.

2. Single-file Implementations

Despite the many features modular DRL libraries offer, understanding all the relevant code of an algorithm is a non-trivial effort. As an example, running the PPO model in Atari games using `Stable Baselines 3 (SB3)` with a debugger involves jumping back and forth between 20 python files that comprise 4000+ LOC (Raffin et al., 2021) (Appendix B). This makes it difficult to understand how the algorithm works due to the sheer amount of code and its complex structure. This is a problem because even small implementation details can have a large impact on the performance of deep RL algorithms (Engstrom et al., 2020), and understanding them has become increasingly important.

`CleanRL` makes it much easier to understand implementation details with a simple idea — putting all implementation details of an algorithm variant into a single file. We call this practice "single-file implementations." Single-file implementations allow us to focus on implementing a specific variant without worrying about handling special cases. Also, for utilities that are not relevant to the algorithm itself, like logging and plotting, we import third-part libraries. As a result, CleanRL produces a codebase with an order of magnitude fewer LOC for each algorithm variant. For example, we have a:

1. `ppo.py` (321 LOC) for the classic control environments, such as `CartPole-v1`,
2. `ppo_atari.py` (337 LOC) for the Atari environments (Bellemare et al., 2013),
3. `ppo_continuous_action.py` (331 LOC) for the robotics environments (e.g., MuJoCo, PyBullet) with continuous action spaces (Schulman et al., 2017).

The single-file implementations have the following benefits.

Transparent learning experience It becomes easier to recognize all aspects of the code in one place. By looking at `ppo.py`, it is straightforward to recognize the core implementation details of PPO. It also becomes easier to identify the difference between algorithm variants via `filediff`. For example, comparing `ppo.py` with `ppo_atari.py` shows a 30 LOC difference required to add environment prepossessing and modify neural networks. Meanwhile, another comparison with `ppo_continuous_action.py` shows a 25 LOC difference required to use normalization and account for continuous action space. See Figure 1 as an example. Being able to display the variant's differences explicitly has helped us explain 37 implementation details of PPO (Huang et al., 2022).

Better debug interactivity Everything is located in a single file, so when debugging, the user does not need to browse different modules like in modular libraries. Additionally, most variables in the files exist in the *global Python name scope*. This means the researchers can use `Ctrl+C` to stop the program execution and check most variables and their shapes in the interactive shell (Appendix C). This is more convenient than using the Python's debugger, which only shows the variables in a specific name scope like in a function.

Painless performance attribution If a new version of our algorithm has obtained a higher performance, we know the exact single file which is responsible for the performance improvement. To attribute the performance improvement, we can simply do a filediff between the current and past versions, and every line of code change is made explicit to us. In comparison, two different versions of modular RL libraries usually involve dozens of file changes, which are more difficult to compare.

Faster prototyping experience CleanRL gives researchers fine-grained control to everything related to the algorithm in a single file, hence making it efficient to develop prototypes without having to subclass like in other modular RL libraries. As an example, invalid action masking (Huang and Ontañón, 2022) is a common technique used in games with large, parameterized action spaces. With `CleanRL`, it takes about 40 LOC to implement (Huang et al., 2022, Sec. 4), whereas in other libraries it could take substantially more LOC (e.g., more than 600 LOC, excluding the test cases[1]) because of overhead such as re-factoring the functional arguments and making more general classes.

Because of these benefits, we have also implemented single-file implementations for Deep Q-learning (Mnih et al., 2013), Categorical Deep Q-learning (Bellemare et al., 2017), Deep Deterministic Policy Gradient (Lillicrap et al., 2016), Twin-delayed Deep Deterministic Policy Gradient (Fujimoto et al., 2018), Soft Actor-cirtic (Haarnoja et al., 2018a), Phasic Policy Gradient (Cobbe et al., 2021), and Random Network Distillation (Burda et al., 2019).

Despite of these benefits of single-file implementations, one downside is the excessive amount of duplicate code. To help reduce the maintenance overhead, we have adopted a series of developmental tools to automatically format code, pin dependencies, scale experiments with cloud providers, etc (Appendix D).

3. Documentation and Benchmark

All CleanRL's single-file implementations are thoroughly documented and benchmarked in our main documentation site (`https://docs.cleanrl.dev/`). For each single-file implementation, we document the original paper and relevant information, usage, an explanation of logged metrics, note-worthy implementation details, and benchmark results which include learning curves, a table comparing performance against reputable sources when applicable, and links to the tracked experiments. In particular, the benchmark experiments are tracked with Weights and Biases (Biewald, 2020), which allows the users to interactively explore other tracked data such as system metrics, hyperparameters, and the agents' gameplay videos. For convenience, we have included tables comparing the performance of CleanRL's single-file implementations against reputable sources when applicable (Appendix A).

4. When to Use CleanRL

CleanRL has its own set of pros and cons like other popular modular RL libraries. For example, modular DRL libraries, such as SB3, offer a friendly end user API — if an end user does not know much about DRL but wants to apply PPO in their tasks, SB3 would be a great fit. Among many other benefits, SB3 makes it easy to configure different components. CleanRL does not have a friendly end user API like `agent.learn()`, but it exposes all implementation details and is easy to read, debug, modify for research, and study RL. Comparatively, CleanRL is well-suited for researchers who need to understand all implementation details of DRL algorithms, and prototype novel features quickly.

CleanRL complements the DRL research community with a unique developing experience. In fact, there is a win-win situation for CleanRL and SB3: "prototype with CleanRL and port to SB3 for wider adoption in the community." CleanRL's codebase often allows researchers to prototype specialized features much quicker. As shown above, the invalid action masking technique with PPO takes ∼40 LOC to implement. Once we have rigorously validated this technique, our results and analysis will provide concrete guidance for porting this technique to SB3, which enable our technique to reach a wider range of audience given SB3's friendly end user APIs.

1. See `https://github.com/Stable-Baselines-Team/stable-baselines3-contrib/pull/25`.

Acknowledgments

We thank Santiago Ontañón for providing helpful feedback and Angela Steyn for proofreading the paper. We thank Google's TPU Research Cloud (TRC) for supporting TPU related experiments. Also, we thank the following people for their open-source contributions:

1. @adamcakg: refactor class arguments (#34)
2. Jordan Terry: fix the `bibtex` entry (#44).
3. Michael Lu: fix links in the docs (#48)
4. Felipe Martins: fix links in the docs (#54)
5. @chutaklee: refactor TD3 class initialization (#37)
6. Ram Rachum: use correct python exception clause (#205)
7. Alexander Nikulin: fix reward normalization wrapper (#209)
8. Ben Trevett: refactor replay buffer (#36)
9. @ElliotMunro200: specify correct python version requirement (#177)
10. Helge Spieker: fix typos (#45)

Appendix A. Benchmark experiments

Like mentioned in the Section 3, we rigorously benchmark our single-file implementations to validate their quality. Below are the tables that compare performance against reputable resources when applicable, where the reported numbers are the final average episodic returns of at least 3 random seeds. For more detailed information, see the main documentation site (`https://docs.cleanrl.dev/`).

A.1 Proximal Policy Optimization Variants and Performance

The following table reports the final episodic returns obtained by the agent in Gym's classic control tasks (Brockman et al., 2016):

Environment	`ppo.py`	`openai/baselies`' PPO (Huang et al., 2022)
CartPole-v1	492.40 ± 13.05	497.54 ± 4.02
Acrobot-v1	-89.93 ± 6.34	-81.82 ± 5.58
MountainCar-v0	-200.00 ± 0.00	-200.00 ± 0.00

The following tables report the final episodic returns obtained by the agent in Gym's Atari tasks (Brockman et al., 2016; Bellemare et al., 2013):

Environment	`ppo_atari.py`	`openai/baselies`' PPO (Huang et al., 2022)[^1]
BreakoutNoFrameskip-v4	416.31 ± 43.92	406.57 ± 31.554
PongNoFrameskip-v4	20.59 ± 0.35	20.512 ± 0.50
BeamRiderNoFrameskip-v4	2445.38 ± 528.91	2642.97 ± 670.37

Environment	`ppo_atari_lstm.py`	`openai/baselies`' PPO (Huang et al., 2022)
BreakoutNoFrameskip-v4	128.92 ± 31.10	138.98 ± 50.76
PongNoFrameskip-v4	19.78 ± 1.58	19.79 ± 0.67
BeamRiderNoFrameskip-v4	1536.20 ± 612.21	1591.68 ± 372.95

Environment	`ppo_atari_multigpu.py` (160 mins)	`ppo_atari.py` (215 mins)
BreakoutNoFrameskip-v4	429.06 ± 52.09	416.31 ± 43.92
PongNoFrameskip-v4	20.40 ± 0.46	20.59 ± 0.35
BeamRiderNoFrameskip-v4	2454.54 ± 740.49	2445.38 ± 528.91

The following table reports the final episodic returns obtained by the agent in Gym's MuJoCo tasks (Brockman et al., 2016; Todorov et al., 2012):

Environment	`ppo_continuous_action.py`	`openai/baselies`' PPO (Huang et al., 2022)
Hopper-v2	2231.12 ± 656.72	2518.95 ± 850.46
Walker2d-v2	3050.09 ± 1136.21	3208.08 ± 1264.37
HalfCheetah-v2	1822.82 ± 928.11	2152.26 ± 1159.84

The following table reports the final episodic returns obtained by the agent in EnvPool's Atari tasks (Brockman et al., 2016; Bellemare et al., 2013; Weng et al., 2022):

Environment	`ppo_atari_envpool.py` (80 mins)	`ppo_atari.py` (220 mins)
Breakout	389.57 ± 29.62	416.31 ± 43.92
Pong	20.55 ± 0.37	20.59 ± 0.35
BeamRider	2039.83 ± 1146.62	2445.38 ± 528.91

The following table reports the final episodic returns obtained by the agent in Procgen tasks (Cobbe et al., 2020):

Environment	`ppo_procgen.py`	`openai/baselies`' PPO (Huang et al., 2022)
StarPilot	31.40 ± 11.73	33.97 ± 7.86
BossFight	9.09 ± 2.35	9.35 ± 2.04
BigFish	21.44 ± 6.73	20.06 ± 5.34

The following table reports the final episodic returns obtained by the agent in Isaac Gym (Makoviychuk et al., 2021):

Environment	`ppo_continuous_action_isaacgym.py` (160 mins)	`Denys88/rl_games` (215 mins)
Cartpole (40s)	413.66 ± 120.93	417.49 (30s)
Ant (240s)	3953.30 ± 667.086	5873.05
Humanoid (350s)	2987.95 ± 257.60	6254.73
Anymal (317s)	29.34 ± 17.80	62.76
BallBalance (160s)	161.92 ± 89.20	319.76
AllegroHand (200m)	762.93 ± 427.92	3479.85
ShadowHand (130m)	427.16 ± 161.79	5713.74

The following table reports the final *episodic length* instead of *episodic return* obtained by the agent in PettingZoo (Terry et al., 2021):

Environment	`ppo_pettingzoo_ma_atari.py` (160 mins)
pong_v3	4153.60 ± 190.80
surround_v2	3055.33 ± 223.68
tennis_v3	14538.02 ± 7005.54

A.2 Deep Deterministic Policy Gradient Variants and Performance

The following tables report the final episodic returns obtained by the agent in Gym's MuJoCo tasks (Brockman et al., 2016; Todorov et al., 2012):

Environment	`ddpg_continuous_action.py`	`OurDDPG.py` (Fujimoto et al., 2018, Tab. 1)	`DDPG.py` using settings from (Lillicrap et al., 2016) in (Fujimoto et al., 2018, Tab. 1)
HalfCheetah	9382.32 ± 1395.52	8577.29	3305.60
Walker2d	1598.35 ± 862.66	3098.11	1843.85
Hopper	1313.43 ± 684.46	1860.02	2020.46

Environment	`ddpg_continuous_action_jax.py` (RTX 3060)	`ddpg_continuous_action_jax.py` (VM w/ TPU)
HalfCheetah	9910.53 ± 673.49	9790.72 ± 1494.85
Walker2d	1397.60 ± 677.12	1314.83 ± 689.71
Hopper	1603.5 ± 727.281	1602.20 ± 696.11

	`ddpg_continuous_action.py` (RTX 2060)
HalfCheetah	9382.32 ± 1395.52
Walker2d	1598.35 ± 862
Hopper	1313.43 ± 684.46

A.3 Twin-Delayed Deep Deterministic Policy Gradient Variants and Performance

The following tables report the final episodic returns obtained by the agent in Gym's MuJoCo tasks (Brockman et al., 2016; Todorov et al., 2012):

Environment	`td3_continuous_action.py`	`TD3.py` (Fujimoto et al., 2018, Tab. 1)
HalfCheetah	9018.31 ± 1078.31	9636.95 ± 859.065
Walker2d	4246.07 ± 1210.84	4682.82 ± 539.64
Hopper	3391.78 ± 232.21	3564.07 ± 114.74

Environment	`td3_continuous_action_jax.py` (RTX 3060)	`td3_continuous_action_jax.py` (VM w/ TPU)
HalfCheetah	9099.93 ± 1171.83	9127.81 ± 965.42
Walker2d	2874.39 ± 1684.57	3519.38 ± 368.02
Hopper	3382.66 ± 242.52	3126.40 ± 558.93

	`td3_continuous_action.py` (RTX 2060)
HalfCheetah	9018.31 ± 1078.31
Walker2d	4246.07 ± 1210.84
Hopper	3391.78 ± 232.21

A.4 Soft Actor-Critic Variant and Performance

The following table reports the final episodic returns obtained by the agent in Gym's MuJoCo tasks (Brockman et al., 2016; Todorov et al., 2012):

Environment	`sac_continuous_action.py`	Haarnoja et al. (2018b)
HalfCheetah-v2	10310.37 ± 1873.21	∼11,250
Walker2d-v2	4418.15 ± 592.82	∼4,800
Hopper-v2	2685.76 ± 762.16	∼3,250

A.5 Phasic Policy Gradient Variant and Performance

The following table reports the final episodic returns obtained by the agent in Procgen tasks (Cobbe et al., 2020):

Environment	`ppg_procgen.py`	`ppo_procgen.py`	`openai/phasic-policy-gradient`
Starpilot (easy)	35.19 ± 13.07	33.15 ± 11.99	42.01 ± 9.59
Bossfight (easy)	10.34 ± 2.27	9.48 ± 2.42	10.71 ± 2.05
Bigfish (easy)	27.25 ± 7.55	22.21 ± 7.42	15.94 ± 10.80

A.6 Deep Q-learning Variants and Performance

The following tables report the final episodic returns obtained by the agent in Gym's Atari tasks (Brockman et al., 2016; Bellemare et al., 2013):

Environment	`dqn_atari.py` 10M steps	Mnih et al. (2015) 50M steps	Hessel et al. (2018, Fig. 5)
BreakoutNoFrameskip-v4	366.928 ± 39.89	401.2 ± 26.9	~230 (10M steps) ~300 (50M steps)
PongNoFrameskip-v4	20.25 ± 0.41	18.9 ± 1.3	~20 (10M steps) ~20 (50M steps)
BeamRiderNoFrameskip-v4	6673.24 ± 1434.37	6846 ± 1619	~6000 (10M steps) ~7000 (50M steps)

Environment	`dqn_atari_jax.py` 10M steps	`dqn_atari.py` 10M steps	Mnih et al. (2015) 50M steps
BreakoutNoFrameskip-v4	377.82 ± 34.91	366.928 ± 39.89	401.2 ± 26.9
PongNoFrameskip-v4	20.43 ± 0.34	20.25 ± 0.41	18.9 ± 1.3
BeamRiderNoFrameskip-v4	5938.13 ± 955.84	6673.24 ± 1434.37	6846 ± 1619

The following tables report the final episodic returns obtained by the agent in Gym's classic control tasks (Brockman et al., 2016):

Environment	`dqn.py`
CartPole-v1	488.69 ± 16.11
Acrobot-v1	-91.54 ± 7.20
MountainCar-v0	-194.95 ± 8.48

Environment	`dqn_jax.py`	`dqn.py`
CartPole-v1	499.84 ± 0.24	488.69 ± 16.11
Acrobot-v1	-89.17 ± 8.79	-91.54 ± 7.20
MountainCar-v0	-173.71 ± 29.14	-194.95 ± 8.48

A.7 Categorical Deep Q-learning Variants and Performance

The following table reports the final episodic returns obtained by the agent in Gym's Atari tasks (Brockman et al., 2016; Bellemare et al., 2013):

Environment	`c51_atari.py` 10M steps	Bellemare et al. (2013, Fig. 14) 50M steps	Hessel et al. (2018, Fig. 5)
BreakoutNoFrameskip-v4	461.86 ± 69.65	748	~500 (10M steps) ~600 (50M steps)
PongNoFrameskip-v4	19.46 ± 0.70	20.9	~20 (10M steps) ~20 (50M steps)
BeamRiderNoFrameskip-v4	9592.90 ± 2270.15	14,074	~12000 (10M steps) ~14000 (50M steps)

The following table reports the final episodic returns obtained by the agent in Gym's classic control tasks (Brockman et al., 2016):

Environment	`c51.py`
CartPole-v1	481.20 ± 20.53
Acrobot-v1	-87.70 ± 5.52
MountainCar-v0	-166.38 ± 27.94

A.8 Random Network Distillation

The following table reports the final episodic returns obtained by the agent in Gym's Atari tasks (Brockman et al., 2016; Bellemare et al., 2013):

Environment	`ppo_rnd_envpool.py`	Burda et al. (2019)
MontezumaRevenge-v5	7100 (1 seed)	8152 (3 seeds)

Appendix B. Stepping Through Stable-baselines 3 Code with a Debugger

In this section, we attempt to run the following Stable-baselines 3 (v1.5.0)[2] code with a debugger to identify the related modules.

```
from stable_baselines3.common.env_util import make_atari_env
from stable_baselines3.common.vec_env import VecFrameStack
from stable_baselines3 import PPO
env = make_atari_env('PongNoFrameskip-v4', n_envs=4, seed=0)
env = VecFrameStack(env, n_stack=4)
model = PPO('CnnPolicy', env, verbose=1)
model.learn(total_timesteps=25_000)
```

Here is the list of the related python files and their lines of code (LOC):

1. `stable_baselines3/ppo/ppo.py` — 315 LOC, 51 lines of docstring (LOD)
2. `stable_baselines3/common/on_policy_algorithm.py` — 280 LOC, 49 LOD
3. `stable_baselines3/common/base_class.py` — 819 LOC, 231 LOD
4. `stable_baselines3/common/utils.py` — 506 LOC, 195 LOD
5. `stable_baselines3/common/env_util.py` — 157 LOC, 43 LOD
6. `stable_baselines3/common/atari_wrappers.py` — 249 LOC, 84 LOD
7. `stable_baselines3/common/vec_env/__init__.py` — 73 LOC, 24 LOD
8. `stable_baselines3/common/vec_env/dummy_vec_env.py` — 126 LOC, 25 LOD
9. `stable_baselines3/common/vec_env/base_vec_env.py` — 375 LOC, 112 LOD
10. `stable_baselines3/common/vec_env/util.py` — 77 LOC, 31 LOD
11. `stable_baselines3/common/vec_env/vec_frame_stack.py` — 65 LOC, 14 LOD
12. `stable_baselines3/common/vec_env/stacked_observations.py` — 267 LOC, 74 LOD
13. `stable_baselines3/common/preprocessing.py` — 217 LOC, 68 LOD
14. `stable_baselines3/common/buffers.py` — 770 LOC, 183 LOD
15. `stable_baselines3/common/policies.py` — 962 LOC, 336 LOD
16. `stable_baselines3/common/torch_layers.py` — 318 LOC, 97 LOD
17. `stable_baselines3/common/distributions.py` — 700 LOC, 228 LOD

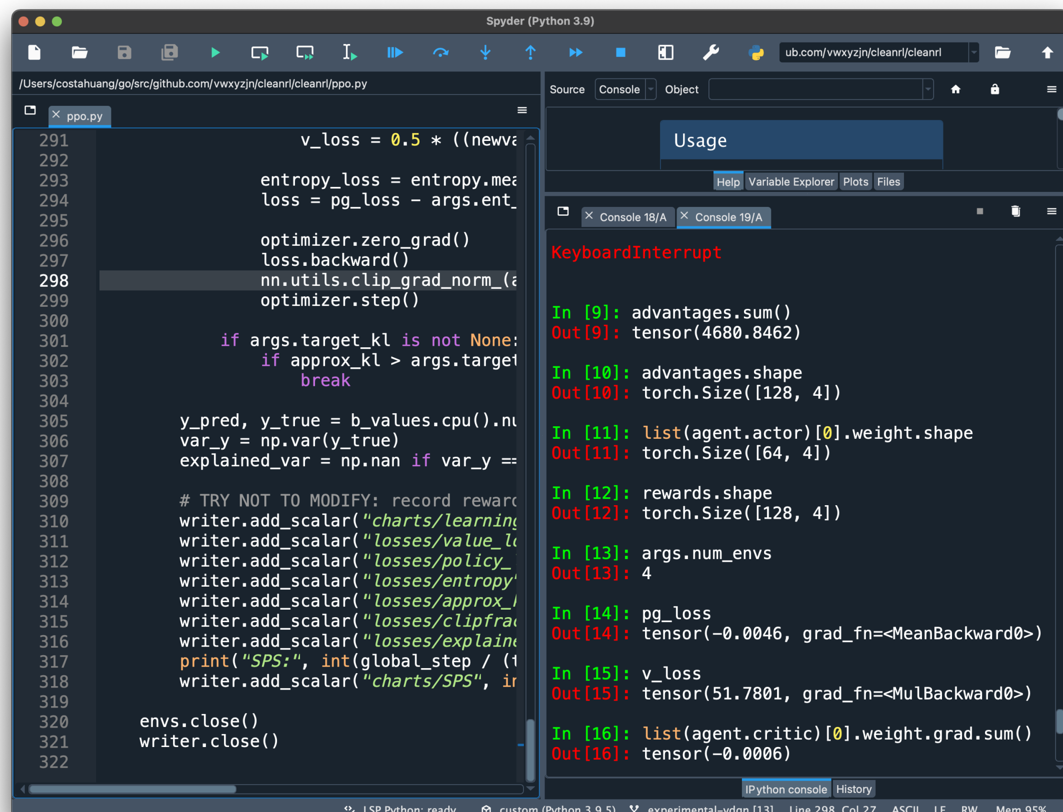
2. https://github.com/DLR-RM/stable-baselines3/releases/tag/v1.5.0

18. `stable_baselines3/common/monitor.py` — 240 LOC, 76 LOD
19. `stable_baselines3/common/logger.py` — 640 LOC, 201 LOD
20. `stable_baselines3/common/callbacks.py` — 603 LOC, 150 LOD

The total LOC involved is 7759. Notice we have labeled the popular utilities such as vectorized environments, Atari environment pre-processing wrappers, and episode statistics recording code with the blue color. This means the total LOC related to core PPO implementation **not** counting the blue color files and lines of docstring is 4498.

Appendix C. Interactive Shell

In `CleanRL`, we have put most of the variables in the *global python name scope*. This makes it easier to inspect the variables and their shapes. The following figure shows a screenshot of the Spyder editor [3], where the code is on the left and the interactive shell is on the right. In the interactive shell, we can easily inspect the variables for debugging purposes without modifying the code.



3. `https://www.spyder-ide.org/`

Appendix D. Maintaining Single-file Implementations

Despite the many benefits that single-file implementations offer, one downside is excessive amount of duplicate code, which makes them difficult to maintain. To help address this challenge, we have adopted a series of development tools to reduce maintenance burden. These tools are:

1. **poetry** (`https://python-poetry.org/`): poetry is a dependency management tool that helps resolve and pins dependency versions. We use poetry to improve reproducibility and provide a smooth dependency installation experience. See our installation documentation (`https://docs.cleanrl.dev/get-started/installation/`) for more detail.

2. **pre-commit** (`https://pre-commit.com/`): pre-commit is a tool that helps us automate a sequence of short tasks (called pre-commit "hooks") such as code formatting. In particular, we always use the following hooks when submitting code to the main repository. See `https://github.com/vwxyzjn/cleanrl/blob/master/CONTRIBUTING.md` for more information.

   (a) **pyupgrade** (`https://github.com/asottile/pyupgrade`): pyupgrade upgrades syntax for newer versions of the language.

   (b) **isort** (`https://github.com/PyCQA/isort`): isort sorts imported dependencies according to their type (e.g, standard library vs third-party library) and name.

   (c) **black** (`https://black.readthedocs.io/en/stable/`): black enforces an uniform code style across the codebase.

   (d) **autoflake** (`https://github.com/PyCQA/autoflake`): autoflake helps remove unused imports and variables.

   (e) **codespell** (`https://github.com/codespell-project/codespell`): codespell helps avoid common incorrect spelling.

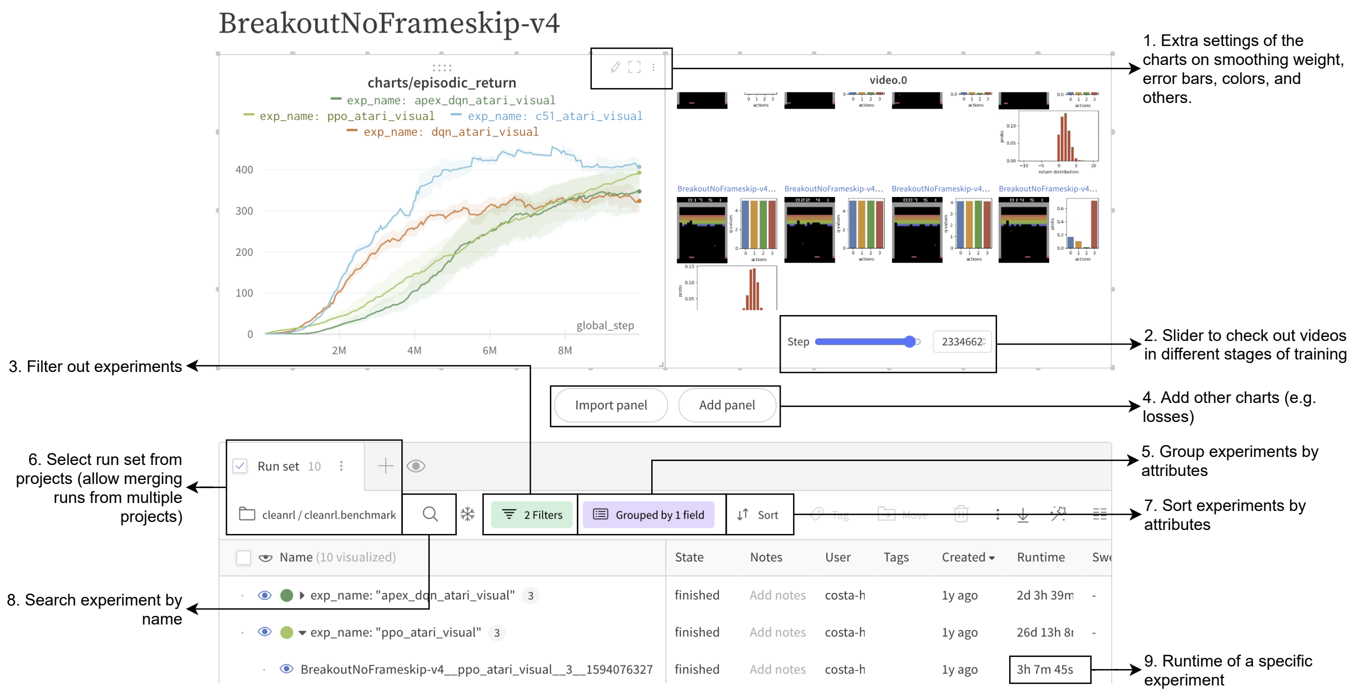
3. **Docker** (`https://www.docker.com/`): docker helps us package the code into a container which can be used to orchestrate training in a reproducible way.

   (a) **AWS Batch** (`https://aws.amazon.com/batch/`): Amazon Web Services Batch could leverage our built containers to run thousands experiments concurrently.

   (b) We have built utilities to help package code into a container and submit to AWS Batch using a few lines of command. In 2020 alone, the authors have run over 50,000+ hours of experiments using this workflow. See `https://docs.cleanrl.dev/cloud/installation/` for more documentation.

Appendix E. W&B Editing Panel

A screenshot of the W&B panel that allows the the users to change smoothing weight, add panels to show different metrics like losses, visualize the videos of the agents' gameplay, filter, group, sort, and search for desired experiments.



Appendix F. Author Contributions

- **Shengyi Huang and Rousslan Fernand Julien Dossa** co-founded CleanRL and has led its overall development.
- **Chang Ye** contributed a prototype with Random Network Distillation (Burda et al., 2019).
- **Shengyi Huang, Rousslan Fernand Julien Dossa, and Chang Ye** are the main code reviewers and maintainers.
- **Jeff Braga** contributed hundreds of hours of tracked experiments in Weights and Biases and submitted various codebase improvements.
- **Dipam Chakraborty** contributed the Phasic Policy Gradient implementation.
- **Kinal Mehta** contributed the Deep Q-learning implementation with JAX.
- **João G.M. Araújo** contributed the Twin-Delayed Deep Deterministic Policy Gradient implementation with JAX.

- **Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, João G.M. Araújo** wrote the paper.

References

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Marc G. Bellemare, Will Dabney, and Rémi Munos. A Distributional Perspective on Reinforcement Learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 449–458. PMLR, 2017. URL `http://proceedings.mlr.press/v70/bellemare17a.html`.

Lukas Biewald. Experiment tracking with weights and biases, 2020. URL `https://www.wandb.com/`. Software available from wandb.com.

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

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=H1lJJnR5Ym`.

Karl Cobbe, Chris Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 2048–2056. PMLR, 13–18 Jul 2020. URL `https://proceedings.mlr.press/v119/cobbe20a.html`.

Karl W Cobbe, Jacob Hilton, Oleg Klimov, and John Schulman. Phasic policy gradient. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 2020–2027. PMLR, 18–24 Jul 2021. URL `https://proceedings.mlr.press/v139/cobbe21a.html`.

Carlo D'Eramo, Davide Tateo, Andrea Bonarini, Marcello Restelli, and Jan Peters. Mushroomrl: Simplifying reinforcement learning research. *Journal of Machine Learning Research*, 2020.

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 *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL `https://openreview.net/forum?id=r1etN1rtPB`.

Scott Fujimoto, Herke van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1582–1591. PMLR, 2018. URL http://proceedings.mlr.press/v80/fujimoto18a.html.

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 Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1856–1865. PMLR, 2018a. URL http://proceedings.mlr.press/v80/haarnoja18b.html.

Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018b.

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018. URL https://ojs.aaai.org/index.php/AAAI/article/view/11796.

Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. In Roman Barták, Fazel Keshtkar, and Michael Franklin, editors, *Proceedings of the Thirty-Fifth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2022, Hutchinson Island, Jensen Beach, Florida, USA, May 15-18, 2022*, 2022. doi: 10.32473/flairs.v35i.130584. URL https://doi.org/10.32473/flairs.v35i.130584.

Shengyi Huang, Rousslan Fernand Julien Dossa, Antonin Raffin, Anssi Kanervisto, and Weixun Wang. The 37 implementation details of proximal policy optimization. In *ICLR Blog Track*, 2022. URL https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.

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 Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 3059–3068. PMLR, 2018. URL http://proceedings.mlr.press/v80/liang18b.html.

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 Yoshua Bengio and Yann LeCun, editors, *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.

Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. Isaac gym: High performance GPU based physics simulation for robot learning. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. URL https://openreview.net/forum?id=fgFBtYgJQX_.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *ArXiv preprint*, abs/1312.5602, 2013. URL https://arxiv.org/abs/1312.5602.

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, Charlie Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *ArXiv preprint*, abs/1707.06347, 2017. URL https://arxiv.org/abs/1707.06347.

J Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis S Santos, Clemens Dieffendahl, Caroline Horsch, Rodrigo Perez-Vicente, et al. Pettingzoo: Gym for multi-agent reinforcement learning. *Advances in Neural Information Processing Systems*, 34:15032–15043, 2021.

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. doi: 10.1109/IROS.2012.6386109.

Jiayi Weng, Huayu Chen, Dong Yan, Kaichao You, Alexis Duburcq, Minghao Zhang, Hang Su, and Jun Zhu. Tianshou: A highly modularized deep reinforcement learning library. *arXiv preprint arXiv:2107.14171*, 2021.

Jiayi Weng, Min Lin, Shengyi Huang, Bo Liu, Denys Makoviichuk, Viktor Makoviychuk, Zichen Liu, Yufan Song, Ting Luo, Yukun Jiang, Zhongwen Xu, and Shuicheng Yan. Envpool: A highly parallel reinforcement learning environment execution engine. *arXiv preprint arXiv:2206.10558*, 2022.