# Efficient Reinforcement Learning Development with RLzoo

Zihan Ding
Imperial College London

Tianyang Yu
Nanchang University

Yanhua Huang
Xiaohongshu Technology Co.

Hongming Zhang
Peking University

Guo Li
Imperial College London

Quancheng Guo
University of Edinburgh

Luo Mai
University of Edinburgh

Hao Dong
Peking University

## ABSTRACT

Many multimedia developers are exploring for adopting Deep Reinforcement Learning (DRL) techniques in their applications. They however often find such an adoption challenging. Existing DRL libraries provide poor support for prototyping DRL agents (i.e., models), customising the agents, and comparing the performance of DRL agents. As a result, the developers often report low efficiency in developing DRL agents. In this paper, we introduce RLzoo, a new DRL library that aims to make the development of DRL agents *efficient.* RLzoo provides developers with (i) high-level yet flexible APIs for prototyping DRL agents, and further customising the agents for best performance, (ii) a model zoo where users can import a wide range of DRL agents and easily compare their performance, and (iii) an algorithm that can automatically construct DRL agents with custom components (which are critical to improve agent's performance in custom applications). Evaluation results show that RLzoo can effectively reduce the development cost of DRL agents, while achieving comparable performance with existing DRL libraries.

**ACM Reference Format:**
Zihan Ding, Tianyang Yu, Yanhua Huang, Hongming Zhang, Guo Li, Quancheng Guo, Luo Mai, and Hao Dong. 2021. Efficient Reinforcement Learning Development with RLzoo. In *Proceedings of Proceedings of the 29th ACM International Conference on Multimedia, Virtual Event, China, October 20–24, 2021 (MM '21),* 5 pages.
https://doi.org/10.1145/3474085.3478323

## 1 INTRODUCTION

Deep reinforcement learning (DRL) has recently become an effective approach to improve the performance of multimedia applications, e.g., gaming [20], self-driving cars [2], language and vision classification [22], and complex optimisation [16]. To adopt DRL techniques, developers often need to construct DRL agents. These agents interact with training environments, e.g., OpenAI Gym [3], RLbench [13], to collect training samples. Samples are sent to DRL algorithms which learn policies, e.g., on-policy algorithms [24, 25] and off-policy algorithms [20], that can maximise agent's performance, i.e., rewards, in interacting with the environments.

Developing a DRL agent that can tackle a real-world application is however challenging. There are several phases that are particularly time-consuming in developing a DRL agent: (i) *Prototyping phase.* A DRL agent contains various components (i.e., environments, DRL algorithms, and training drivers). To create a DRL agent, developers have to spend tremendous time in importing external training environments, modifying the environments to make them compatible with downstream DRL algorithms, and writing training drivers that can iteratively train the agents and distribute the training onto heterogeneous processors [17]. (ii) *Customisation phase.* The default configuration (e.g., neural network architecture) of a DRL algorithm often exhibit sub-optimal performance [13, 16, 22]. Developers thus must customise the DRL algorithm to improve its performance. (iii) *Algorithm comparison phase.* In tackling a training environment, there are often multiple DRL algorithms available [11, 20, 24, 25]. Developers usually need to implement all these algorithms and compare their performance.

Even though several DRL libraries have become available recently, developers still find it inefficient in using these libraries to construct DRL agents for custom applications. On the one hand, tutorial-oriented DRL libraries, such as OpenAI Baselines [6], Stable Baselines and Coach [4], provide command-line-based interfaces and they focus on reproducing classical benchmarks. They do not have low-level APIs which are necessary to control how a DRL agent is being trained, and how it is customised. On the other hand, research-oriented DRL libraries, such as Tianshou [14], keras-rl [23], and Tensorforce [15], provide flexible APIs (e.g., defining the reward functions or policy networks), useful for defining custom DRL agents. They, however, fail to provide expressive high-level APIs to help prototype DRL agents, and access commonly used DRL agents.

In this paper, we introduce RLzoo, a DRL library that can enable developers to efficiently prototype, train and evaluate DRL agents. The design of RLzoo makes several contributions:

**(i) High-level yet flexible APIs for declaring DRL agents.** RLzoo contains high-level APIs for prototyping DRL agents. These APIs contain *expressive functions* for importing external training environments, declaring DRL algorithms, and launching training drivers which can iteratively train the policies and scale the training to distributed nodes.

Yet, RLzoo's APIs do not compromise flexibility. They contain flexible functions which allow DRL agents to take custom agent components, e.g., providing a custom neural network for a DRL

*MM '21, October 20–24, 2021, Virtual Event, China*
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8651-7/21/10…$15.00
https://doi.org/10.1145/3474085.3478323

algorithm or providing a custom communication topology for distributed DRL agents. By consolidating both the high-level and low-level APIs, RLzoo is effective in facilitating both the prototyping phase and the customisation phase in developing DRL agents.

**(ii) DRL model zoo.** RLzoo provides a DRL model zoo to further facilitate the algorithm comparison phase in developing DRL agents. The model zoo contains many useful pre-defined DRL environments and algorithms. Generally, these algorithms can be classified into those for tutorial purposes (i.e., beginners) and those for research purposes (i.e., professionals). In particular, RLzoo puts a focus on offering support for distributed DRL algorithms [8, 11] and robot-learning-related environments, e.g., RLBench [13].

Further, the RLzoo model zoo contains an easy-to-use *agent training notebook*. RLzoo users can track the performance (e.g., reward) and configuration (e.g., hyper-parameters) of DRL agents, and evaluate different DRL agents in an intuitive manner.

**(iii) Automatic algorithm for constructing DRL agents.** RLzoo minimises developer's effort for integrating custom components into DRL agents, or re-configuring the agents for new scenarios. This is achieved by a novel algorithm that can automatically construct DRL agents with various custom components. Specifically, this algorithm has *adaptors* for connecting the components (e.g., environments and DRL algorithms) in DRL agents. The adaptors can automatically infer the input/output shapes of agent components. As long as changes in these components are detected, the adaptors can automatically re-configure the DRL agent, which avoids the need for developers to make manual modification as in existing DRL libraries.

RLzoo is implemented as a Python library based on TensorFlow [1], TensorLayer [7] and KungFu [19]. It is open-sourced on Github[1] in December, 2019. It has attracted numerous users from both education and industry. RLzoo has also been used for implementing demonstrations in a multi-lingual DRL textbook [2].

## 2 RLZOO DESIGN

In this section, we present the design of RLzoo. We first describe the APIs in RLzoo, and then the design of RLzoo model zoo. We will introduce the algorithm for automatically constructing DRL agents, and end this section with presenting the generic system architecture of distributed DRL training in RLzoo.

### 2.1 High-level yet Flexible APIs

The API design of RLzoo has the following goals: (i) We want to enable users to use high-level expressive functions in declaring a DRL agent with custom training environments, DRL algorithms and training drivers; (ii) We want to support users to flexibly customise their DRL agents by plugging different custom objects into DRL implementations.

We introduce the RLzoo APIs using a sample program as shown in Listing 1. To declare a DRL agent, RLzoo users first need to choose an environment (i,e., `Pendulum-v0`) (line 3~5). Based on the chosen environment, users further decide a DRL algorithm: `TD3` (line 6). In order to train this DRL agent, the users obtain its default construction parameters (line 7). The algorithm parameters (`alg_params`) are used for constructing a DRL agent (line 8). This

```
1 from rlzoo.common.env_wrappers import build_env
2 from rlzoo.common.utils import call_default_params
3 env_type = 'classic_control'
4 env_name = 'Pendulum-v0'
5 env = build_env(env_name, env_type) # Build environment
6 from rlzoo.algorithms import TD3 # Choose algorithm
7 alg_params, learn_params = call_default_params(env,
      env_type, 'TD3') # Create configuration
8 agent = TD3(**alg_params) # Construct agent
9 agent.learn(env, 'train', **learn_params) # Launch training
```

**Listing 1: Sample RLzoo program**

agent then launches its training process (line 9) given the environment and hyper-parameters (`learn_params`). As we can see in this program, a DRL agent can be declared with 3 abstracted steps (9 lines of code).

RLzoo allows users to flexibly customise DRL agents. To customise a DRL agent, RLzoo users can use the `call_default_params()`. This function returns an agent's configuration and training hyper-parameters as two custom dictionaries. To provide a custom neural network, users can access the default neural networks through the key '`net_list`' in `alg_params` dictionary. They can replace the default networks with new custom networks. In the same manner, RLzoo users can plug in many custom objects, e.g., optimisers, and hyper-parameters like learning rate and batch size, into DRL agents.

### 2.2 Model Zoo

RLzoo further provides users with a model zoo to import pre-defined DRL agents and evaluate the performance of DRL agents:

**Pre-defined DRL algorithms.** There are numerous DRL algorithms available in RLzoo. Specifically, there are (i) classical DRL algorithms, such as the deep Q-network and its variants in the discrete action spaces, as well as (ii) advanced DRL algorithms, such as hindsight experience replay, deep deterministic policy gradient, twin delayed deep deterministic policy gradient, soft actor-critic, advantage actor-critic, asynchronous advantage actor-critic, proximal policy optimisation, distributed proximal policy optimisation, trust region policy optimisation. All these algorithms come with default effective hyper-parameters that help to reproduce SOTA results reported in their papers.

**Supported training environments.** There are also numerous training environments available in RLzoo. There are typical training environments, such as Atari, Box2d, Classic control, MuJoCo, Robotics in OpenAI Gym, and DeepMind Control Suite. In addition, developers can access to more up-to-date environments such as those used for emerging realistic robot learning, e.g., RLBench [13]. These environments produce complex observations represented as compound dictionaries, and they can be used by practitioners to test DRL with robots.

**Agent training notebook.** RLzoo users can exploit a high-level agent training notebook to manage the configurations of DRL agents, analyse their training performance metrics This notebook is implemented based on the Jupyter Notebook. It tracks the configurations of being evaluated DRL agents and stores the agents' traces for performance analysis. Specifically, the notebook displays

[1]https://github.com/tensorlayer/RLzoo
[2]https://deepreinforcementlearningbook.org/

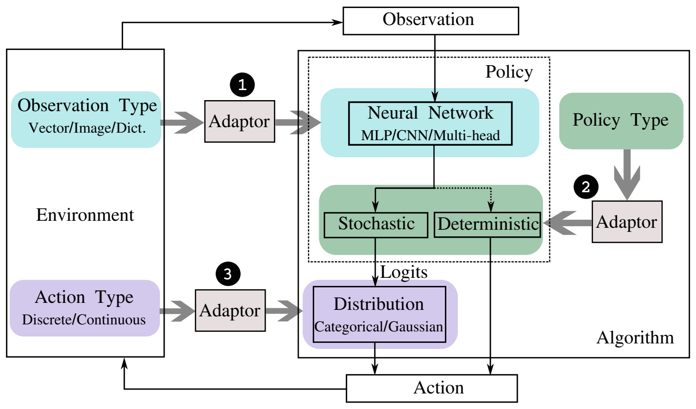

**Figure 1: Automatic construction of a DRL agent.**

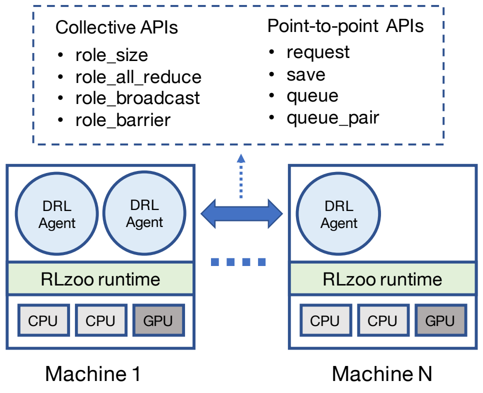

**Figure 2: Distributed trianing architecture of RLzoo agents.**

agent performance metrics, including agent configurations, learning status (e.g., training steps and instant rewards), and training results (e.g., averaged rewards over time and values of loss functions). Based on these metrics, developers can infer the impacts of modification made towards the DRL agents, thus facilitating the tuning of such agents for better performance.

## 2.3 Automatic Agent Construction

RLzoo provides developers with an algorithm to automatically construct custom DRL agents. For those agents, users can plug various user-defined components into existing agent implementation. There must be an approach to automatically adapt the agent implementation based on user-defined components; otherwise, developers have to manually modify the agents whenever a custom component is provided, as in existing DRL libraries, making the development of custom DRL agents tedious and expensive [18].

Figure 1 illustrates the automatic agent construction algorithm in RLzoo. First of all, this algorithm places an *observation adaptor* between the observation from the environment and the neural network (see ❶). This adaptor infers the type of observations and produces different types of neural networks, e.g., a MLP for a vector, the CNN for an image, and a multi-head architecture for a hybrid dictionary. The *policy adaptor* is placed in between the algorithm selection and the policy output (see ❷). Based on the stochastic/deterministic nature of the DRL algorithm, this adaptor produces corresponding policy outputs for each selected DRL algorithm. The *action adaptor* is placed in between the policy (stochastic only) and the environment (see ❸): if the environment requires discrete action, this final adaptor will produce a categorical distribution to represent the action; if the action needs to be continuous, the adaptor produces policy output as a diagonal Gaussian distribution.

## 2.4 Distributed Agent Training

DRL agents usually need to accelerate computation using parallel heterogeneous processors. To achieve this, developers often rely on the native *multiprocess* library in Python. The usage of such a library, however, is generally limited within a single machine. To use distributed machines, developers must use external libraries, such as Ray [21] and Acme [12]. These libraries provide *custom* Remote-Process-Communication (RPC) programming interfaces. Developers must largely modify existing DRL programs in order to adopt these RPC libraries.

In designing RLzoo, we want to minimise developer's efforts in modifying existing single-node programs when scaling out DRL agents. Our idea is to ensure RLzoo's distributed training APIs can follow the convention in the API design of the popular multiprocess library. For each key component in the multiprocess library (e.g., queues and pipes), RLzoo provides *equivalent distributed implementations*. Hence developers can easily replace single-node communication components with those equivalent in RLzoo. Further, RLzoo extends the APIs of the multiprocess library. It provides novel collective communication APIs (e.g., all-reduce) to achieve complex communication patterns among DRL agents [19].

Figure 2 shows the distributed architecture of RLzoo DRL agents. In a cluster, RLzoo replicates *RLzoo runtime* on each machine. This runtime launches RLzoo DRL agents as Python processes, and assigns CPUs and GPUs to the agents. The DRL agents communicate data using expressive communication functions defined in the *collective APIs* and *point-to-point APIs* (Figure 2):

**Agent collective APIs**: RLzoo agents are often assigned with different roles (e.g., actors, learners and inference servers as in IMPALA [8]). The agents in the same role can use (i) *role_all_reduce* to synchronise the gradients among parallel learners, (ii) *role_broadcast* to broadcast weights to parallel learners (which produce gradients to update DRL models), and (iii) *role_barrier* to coordinate the synchronous execution of parallel actors (which collect training trajectories from environments).

**Agent point-to-point APIs**: Developers can use (i) *save* and *request* to asynchronously push and pull weights among DRL agents, and (ii) *queue* or *queue_pair* to exchange data among those agents, similar to the queues in the Python multiprocess library.

## 3 EVALUATION

In this section, we compare RLzoo with other DRL libraries in terms of the supported algorithms, supported environments and their API designs. We choose the following popular libraries as baseline: OpenAI Baselines [6], Tianshou [14], Coach [4], ReAgent [10], garage [9], keras-rl [23], MushroomRL [5] and Tensorforce [15].

**Algorithms.** We first evaluate the algorithm support. As we can see from Table 1, RLzoo supports 12 DRL algorithms, whereas Coach supports 11 algorithms and other libraries support less than 10

| Library | # Algo. | # Env. | Image | Vector | Dict. | LoC |
|---|---|---|---|---|---|---|
| RLzoo | 12 | 7 | ✓ | ✓ | ✓ | 4 |
| Baselines | 9 | 5 | ✓ | ✓ | ✓ | N/A |
| Tianshou | 8 | 5 | ✓ | ✓ | ✓ | 15-20 |
| Coach | 11 | 8 | ✓ | ✓ | ✗ | N/A |
| ReAgent | 4 | 3 | ✓ | ✓ | ✗ | 5 |
| garage | 9 | 6 | ✓ | ✓ | ✗ | 5-10 |
| keras-rl | 3 | 5 | ✓ | ✓ | ✓ | 10-15 |
| MushroomRL | 9 | 7 | ✓ | ✓ | ✗ | 5-10 |
| Tensorforce | 8 | 5 | ✓ | ✓ | ✓ | 5-15 |

**Table 1: Comparison of different DRL libraries.**

algorithms. A key difference between RLzoo and other libraries is its supporting *distributed* DRL algorithms, which makes RLzoo one of the few libraries that supports distributed DRL algorithms such as DPPO. This type of algorithms is increasingly critical because practitioners have recently achieved great success of training DRL agents using parallel learning framework [11].

**Environments.** We then evaluate the environment support. As shown in Table 1, RLzoo supports 7 environments, making it among those libraries, e.g., Coach and MushroomRL, that provide a large collection of environments. A key feature for RLzoo is its support for all observations types (e.g., Vector, Image, and Dictionary). The other library: keras-rl, which can offer the same full support, only provide 3 DRL algorithms, whereas RLzoo can support 12 DRL algorithms. This shows the importance of achieving adaptive agent construction in RLzoo: new observations can be automatically supported by all DRL algorithms. In addition, the full observation support also makes RLzoo the only library, as far as we know, that supports an important environment: RLbench. This environment has growing popularity due to the recent booming of robot learning applications. It produces complex observations that contain images, vectors and dictionaries, making it difficult to be supported by existing libraries.

**API expressiveness.** We evaluate the API design by counting the lines of code (LoC) for declaring DRL agents. We exclude Baselines and Coach because they have only command-line interfaces. The LoCs here only consider necessary code for declaring agents, excluding other lines for importing libraries or assigning values for variables. As we can see in Table 1, RLzoo requires 4 LoCs to declare DRL agent while the ReAgent library comes as the second, costing 5 LoCs on average. Other programmable DRL libraries require users to write around 10 - 20 LoCs. In addition, RLzoo differentiates with other libraries in terms of its support for customising agents. This makes RLzoo an attractive option for robot learning users who often need to (i) deal with RGB-D camera produced by the learning environment: RLBench, and (ii) adopt customised network architectures like recurrent layers.

## 4 CONCLUSION

This paper introduces RLzoo, a novel DRL library that makes the development of DRL agents efficient. RLzoo provides high-level yet flexible APIs for declaring DRL agents. These APIs are particularly efficient in prototyping DRL agents, and scaling out the training of agents to many nodes. RLzoo further comes with a model zoo, enabling developers to easily evaluate different DRL algorithms. In the future, we will consistently improve the API design of RLzoo, e.g., providing better support for implementing emerging multi-agent DRL algorithms. We will also add new DRL algorithms into the model zoo, especially those targeting robot learning.

## REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] A. Amini, I. Gilitschenski, J. Phillips, J. Moseyko, R. Banerjee, S. Karaman, and D. Rus. 2020. Learning Robust Control Policies for End-to-End Autonomous Driving From Data-Driven Simulation. *IEEE Robotics and Automation Letters* 5, 2 (2020), 1143–1150.

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

[4] Itai Caspi, Gal Leibovich, Gal Novik, and Shadi Endrawis. 2017. Reinforcement Learning Coach. (Dec. 2017). https://doi.org/10.5281/zenodo.1134899

[5] Carlo D'Eramo, Davide Tateo, Andrea Bonarini, Marcello Restelli, and Jan Peters. 2020. MushroomRL: Simplifying Reinforcement Learning Research. https://github.com/MushroomRL/mushroom-rl. (2020).

[6] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. 2017. OpenAI Baselines. https://github.com/openai/baselines. (2017).

[7] Hao Dong, Akara Supratak, Luo Mai, Fangde Liu, Axel Oehmichen, Simiao Yu, and Yike Guo. 2017. Tensorlayer: a versatile library for efficient deep learning development. In *Proceedings of the 25th ACM international conference on Multimedia*. 1201–1204.

[8] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. 2018. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning*. PMLR, 1407–1416.

[9] The garage contributors. 2019. Garage: A toolkit for reproducible reinforcement learning research. https://github.com/rlworkgroup/garage. (2019).

[10] Jason Gauci, Edoardo Conti, Yitao Liang, Kittipat Virochsiri, Zhengxing Chen, Yuchen He, Zachary Kaden, Vivek Narayanan, and Xiaohui Ye. 2018. Horizon: Facebook's Open Source Applied Reinforcement Learning Platform. *arXiv preprint arXiv:1811.00260* (2018).

[11] Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, SM Eslami, et al. 2017. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286* (2017).

[12] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, et al. 2020. Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979* (2020).

[13] Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J Davison. 2020. Rlbench: The robot learning benchmark & learning environment. *IEEE Robotics and Automation Letters* 5, 2 (2020), 3019–3026.

[14] Dong Yan Hang Su Jun Zhu Jiayi Weng, Minghao Zhang. 2020. Tianshou. https://github.com/thu-ml/tianshou. (2020).

[15] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. 2017. Tensorforce: a TensorFlow library for applied reinforcement learning. Web page. (2017). https://github.com/tensorforce/tensorforce

[16] Ke Li and Jitendra Malik. 2017. Learning to optimize neural nets. *arXiv preprint arXiv:1703.00441* (2017).

[17] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2018. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*. PMLR, 3053–3062.

[18] Luo Mai, Alexandros Koliousis, Guo Li, Andrei-Octavian Brabete, and Peter Pietzuch. 2019. Taming hyper-parameters in deep learning systems. *ACM SIGOPS Operating Systems Review* 53, 1 (2019), 52–58.

[19] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. 2020. KungFu: Making Training in Distributed Machine Learning Adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 937–954.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[21] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 561–577.

[22] Jongchan Park, Joon-Young Lee, Donggeun Yoo, and In So Kweon. 2018. Distort-and-recover: Color enhancement using deep reinforcement learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5928–5936.

[23] Matthias Plappert. 2016. keras-rl. https://github.com/keras-rl/keras-rl. (2016).

[24] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust region policy optimization. In *International conference on machine learning*. 1889–1897.

[25] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8, 3-4 (1992), 229–256.