Journal of Machine Learning Research 22 (2021) 1-14 Submitted 4/20; Revised 12/20; Published 4/21

ChainerRL: A Deep Reinforcement Learning Library

Yasuhiro Fujita fujita@preferred.jp
Prabhat Nagarajan prabhat@preferred.jp
Toshiki Kataoka kataoka@preferred.jp
Preferred Networks
Tokyo, Japan

Takahiro Ishikawa sykwer@g.ecc.u-tokyo.ac.jp
The University of Tokyo
Tokyo, Japan

Editor: Andreas Mueller

Abstract

In this paper, we introduce `ChainerRL`, an open-source deep reinforcement learning (DRL) library built using Python and the `Chainer` deep learning framework. `ChainerRL` implements a comprehensive set of DRL algorithms and techniques drawn from state-of-the-art research in the field. To foster reproducible research, and for instructional purposes, `ChainerRL` provides scripts that closely replicate the original papers' experimental settings and reproduce published benchmark results for several algorithms. Lastly, `ChainerRL` offers a visualization tool that enables the qualitative inspection of trained agents. The `ChainerRL` source code can be found on GitHub: `https://github.com/chainer/chainerrl`.

Keywords: reinforcement learning, deep reinforcement learning, reproducibility, open source software, chainer

1. Introduction

Since its resurgence in 2013 (Mnih et al., 2013), deep reinforcement learning (DRL) has undergone tremendous progress, and has enabled significant advances in numerous complex sequential decision-making problems (Mnih et al., 2015; Silver et al., 2018; Levine et al., 2016; Kalashnikov et al., 2018). The machine learning community has witnessed a growing body of literature on DRL algorithms (Henderson et al., 2018). However, coinciding with this rapid growth has been a growing concern about the state of reproducibility in DRL (Henderson et al., 2018). The growing body of algorithms and increased reproducibility concerns beget the need for comprehensive libraries, tools, and implementations that can aid RL-based research and development.

Many libraries aim to address these challenges in different ways. `rllab` (Duan et al., 2016) and its successor, `garage`, provide systematic benchmarking of continuous-action algorithms on their own benchmark environments. `Dopamine` (Castro et al., 2018) primarily focuses on DQN and its extensions for discrete-action environments. `rlpyt` (Stooke and Abbeel, 2019) supports both discrete and continuous-action algorithms from the three classes: policy gradient (with V-functions), deep Q-learning, and policy gradient with Q-functions. Other libraries also support diverse sets of algorithms (Dhariwal et al., 2017; Caspi et al., 2017; Hill et al., 2018; Liang et al., 2018). `catalyst.RL` (Kolesnikov and Hrinchuk, 2019) aims

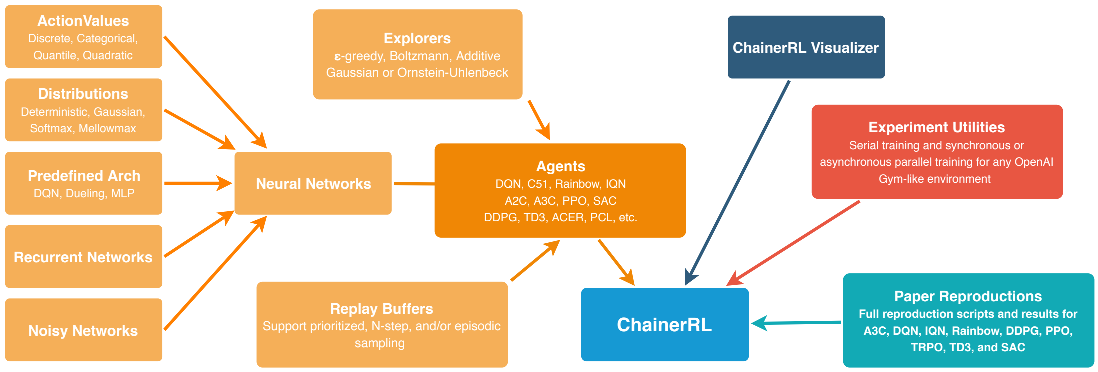

Figure 1: A depiction of `ChainerRL`. Using `ChainerRL`'s building blocks, DRL algorithms, called agents, are written by implementing the `Agent` interface. Agents can be trained with the experiment utilities and inspected with the `ChainerRL Visualizer`.

to address reproducibility issues in RL via deterministic evaluations and by tracking code changes for continuous-action algorithms.

In this paper, we introduce `ChainerRL`, an open-source Python DRL library supporting both CPU and GPU training, built off of the `Chainer` (Tokui et al., 2019) deep learning framework. `ChainerRL` offers a comprehensive set of algorithms and abstractions, a set of "reproducibility scripts" that replicate research papers, and a companion visualizer to inspect agents.

2. Design of ChainerRL

In this section, we describe `ChainerRL`'s design, as in Figure 1.

2.1 Agents

In `ChainerRL`, each DRL algorithm is written as a class that implements the `Agent` interface. The `Agent` interface provides a mechanism through which an agent interacts with an environment, e.g., through an abstract method `Agent.act_and_train(obs, reward, done)` that takes as input the current observation, the previous step's immediate reward, and a flag for episode termination, and returns the agent's action to execute in the environment. By implementing such methods, both the update rule and the action-selection procedure are specified for an algorithm.

An agent's internals consist of any model parameters needed for decision-making and model updating. `ChainerRL` includes several built-in agents that implement key algorithms including the DQN (Mnih et al., 2015) family of algorithms, as well as several policy gradient and actor-critic algorithms.[1]

1. `ChainerRL`'s algorithms include: DQN (Mnih et al., 2015), Double DQN (Van Hasselt et al., 2016), Categorical DQN (Bellemare et al., 2017), Rainbow (Hessel et al., 2017), Implicit Quantile Networks (IQN) (Dabney et al., 2018), Off-policy SARSA, (Persistent) Advantage Learning (Bellemare et al.,

2.2 Experiments

While users can directly interact with agents, `ChainerRL` provides an `experiments` module that manages agent-environment interactions as well as training/evaluation schedules. This module supports any environment that is compatible with `OpenAI Gym`'s `Env` (Brockman et al., 2016). An experiment takes as input an agent and an environment, queries the agent for actions, executes them in the environment, and feeds the agent the rewards for training updates. Moreover, an experiment can periodically perform evaluations and collect evaluation statistics. Through the `experiments` module, `ChainerRL` supports batch or asynchronous training, enabling agents to act, train, and evaluate synchronously or asynchronously in several environments in parallel. A full list of synchronous and asynchronous agents is provided in the appendix.

2.3 Developing a New Agent

The `Agent` interface is defined very abstractly and flexibly so that users can easily implement new algorithms while leveraging the `experiments` utility and parallel training infrastructure. To develop a new agent, we first create a class that inherits `Agent`. Next, the learning update rules and the agent's action-selection mechanisms are implemented using `ChainerRL`'s provided building blocks (see Section 2.4). Once an agent is created, the agent and a `Gym`-like environment can be given to the `experiments` module to easily train and evaluate the agent within the specified environment.

2.4 Agent Building Blocks

`ChainerRL` offers a set of reusable components for building new agents, including `ChainerRL`'s built-in agents. Though not comprehensive, we highlight here some of the building blocks that demonstrate the flexibility and reusability of `ChainerRL`.

Explorers For building action-selection mechanisms during training, `ChainerRL` has built-in explorers including ϵ-greedy, Boltzmann exploration, additive Gaussian noise, and additive Ornstein-Uhlenbeck noise (Lillicrap et al., 2016).

Replay buffers Replay buffers (Lin, 1992; Mnih et al., 2015) have become standard tools in off-policy DRL. `ChainerRL` supports traditional uniform-sampling replay buffers, episodic buffers for sampling past (sub-)episodes for recurrent models, and prioritized buffers that prioritize sampled transitions (Schaul et al., 2016). `ChainerRL` also supports sampling N steps of transitions, for algorithms based on N-step returns.

Neural networks While `ChainerRL` supports any `Chainer` model, it has several pre-defined architectures, including DQN architectures, dueling network architectures (Wang et al., 2016), noisy networks (Fortunato et al., 2018), and multi-layer perceptrons. Recurrent models are supported for many algorithms, including DQN and IQN.

2016), (Asynchronous) Advantage Actor-Critic (A2C (Wu et al., 2017), A3C (Mnih et al., 2016)), Actor-Critic with Experience Replay (ACER) (Wang et al., 2017), Deep Deterministic Policy Gradients (DDPG) (Lillicrap et al., 2016), Twin-delayed double DDPG (TD3) (Fujimoto et al., 2018), Proximal Policy Optimization (PPO) (Schulman et al., 2017), REINFORCE (Williams, 1992), Trust Region Policy Optimization (TRPO) (Schulman et al., 2015), and Soft Actor-Critic (SAC) (Haarnoja et al., 2018).

Distributions `Distribution`s are parameterized objects for modeling action distributions. Network models that return `Distribution` objects are considered policies. Supported policies include Gaussian, Softmax, Mellowmax (Asadi and Littman, 2017), and deterministic policies.

Action values Similar to `Distributions`, `ActionValue`s parameterizing the values of actions are used as outputs of neural networks to model Q-functions. Supported Q-functions include the standard discrete-action Q-function typical of DQN as well as categorical (Bellemare et al., 2017) and quantile (Dabney et al., 2018) Q-functions for distributional RL. For continuous action spaces, quadratic Q-functions called Normalized Advantage Functions (NAFs) (Gu et al., 2016) are also supported.

By combining these agent building blocks, users can easily construct complex agents such as Rainbow (Hessel et al., 2017), which combines six features into a single agent. This ability is highlighted in Appendix D, which provides a pseudocode construction of a Rainbow agent and trains it in multiple parallel environments in just a few lines.

2.5 Visualization

`ChainerRL` is accompanied by the `ChainerRL Visualizer`, which takes as input an environment and an agent, and enables users to easily inspect agents from a browser UI. With the visualizer, one can visualize the portions of the pixel input that the agent is attending to as a saliency map (Greydanus et al., 2018). Additionally, users can either manually step through the episode or view full rollouts of agents. Moreover, the visualizer depicts the probabilities with which the agent will perform specific actions. If the agent learns Q-values or a distribution of Q-values, the predicted Q-value or Q-value distribution for each action can be displayed. Figure 2 in Appendix C depicts some of these features.

3. Reproducibility

Many DRL libraries offer implementations of algorithms but often deviate from the original paper's implementation details. We provide a set of "reproducibility scripts", which are compact examples (i.e., single files) of paper implementations written with `ChainerRL` that match, as closely as possible, the original paper's (or in some cases, another published paper's) implementation and evaluation details. `ChainerRL` currently has "reproducibility scripts" for DQN, IQN, Rainbow, A3C, DDPG, TRPO, PPO, TD3, and SAC. For each of these algorithms and domains, we have released pretrained models for every domain, totaling hundreds of models. Moreover, for each script, we provide full tables of our scores and compare them against scores reported in the literature (Tables 2 and 4 in Appendix B).

4. Conclusion

This paper introduced `ChainerRL` and the `ChainerRL Visualizer`. `ChainerRL`'s comprehensive suite of algorithms, flexible APIs, visualization tools, and faithful reproductions can accelerate the research and application of DRL algorithms. While `ChainerRL` targets Chainer users, we have developed an analogous library, `PFRL`, for PyTorch users.[2]

2. The `PFRL` code is located at `https://github.com/pfnet/pfrl`.

Acknowledgments

We thank Avinash Ummadisingu, Mario Ynocente Castro, Keisuke Nakata, Lester James V. Miranda, and all the open source contributors for their contributions to the development of ChainerRL. We thank Kohei Hayashi and Jason Naradowsky for useful comments on how to improve the paper. We thank the many authors who fielded our questions when reproducing their papers, especially George Ostrovski.

References

Kavosh Asadi and Michael L. Littman. An Alternative Softmax Operator for Reinforcement Learning. In *ICML*, 2017.

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.

Marc G. Bellemare, Georg Ostrovski, Arthur Guez, Philip S. Thomas, and Rémi Munos. Increasing the Action Gap: New Operators for Reinforcement Learning. In *AAAI*, 2016.

Marc G. Bellemare, Will Dabney, and Rémi Munos. A Distributional Perspective on Reinforcement Learning. In *ICML*, 2017.

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

Itai Caspi, Gal Leibovich, Gal Novik, and Shadi Endrawis. Reinforcement learning coach, December 2017. URL `https://doi.org/10.5281/zenodo.1134899`.

Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G. Bellemare. Dopamine: A Research Framework for Deep Reinforcement Learning. 2018.

Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit Quantile Networks for Distributional Reinforcement Learning. In *ICML*, 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 *ICML*, 2016.

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy Networks for Exploration. In *ICLR*, 2018.

Scott Fujimoto, Herke van Hoof, and Dave Meger. Addressing Function Approximation Error in Actor-Critic Methods. In *ICML*, 2018.

Sam Greydanus, Anurag Koul, Jonathan Dodge, and Alan Fern. Visualizing and Understanding Atari Agents. In *ICML*, 2018.

Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous Deep Q-Learning with Model-based Acceleration. In *ICML*, 2016.

Tuomas Haarnoja, Henry Zhu, George Tucker, and Pieter Abbeel. Soft Actor-Critic Algorithms and Applications. *arXiv preprint arxiv:1812.05905*, 2018.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep Reinforcement Learning that Matters. In *AAAI*, 2018.

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. In *AAAI*, 2017.

Ashley Hill, Antonin Raffin, Maximilian Ernestus, 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.

Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, and Sergey Levine. QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation. In *CoRL*, 2018.

Sergey Kolesnikov and Oleksii Hrinchuk. Catalyst.RL: A Distributed Framework for Reproducible RL Research. *arXiv preprint arXiv:1903.00027*, 2019.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-End Training of Deep Visuomotor Policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. RLlib: Abstractions for Distributed Reinforcement Learning. In *ICML*, 2018.

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 *ICLR*, 2016.

Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3-4):293–321, 1992.

Marlos C Machado, Marc G Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents. *Journal of Artificial Intelligence Research*, 61: 523–562, 2018.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei a Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. ISSN 0028-0836. URL `http://dx.doi.org/10.1038/nature14236`.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *ICML*, 2016.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay. In *ICLR*, 2016.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust Region Policy Optimization. In *ICML*, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

Adam Stooke and Pieter Abbeel. rlpyt: A Research Code Base for Deep Reinforcement Learning in PyTorch. *arXiv preprint arxiv:1909.01500*, 2019.

Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. Chainer: A Deep Learning Framework for Accelerating the Research Cycle. In *KDD*, 2019.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-learning. In *AAAI*, 2016.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling Network Architectures for Deep Reinforcement Learning. In *ICML*, 2016.

Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample Efficient Actor-Critic with Experience Replay. In *ICLR*, 2017.

RJ Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8(3-4):229–256, 1992.

Yuhuai Wu, Elman Mansimov, Shun Liao, Alec Radford, and John Schulman. OpenAI Baselines: ACKTR & A2C. `https://openai.com/blog/baselines-acktr-a2c/`, 2017.

Appendix A. Agents

`ChainerRL` implements several kinds of agents, supporting discrete-action agents, continuous-action agents, recurrent agents, batch agents, and asynchronous agents. Asynchronous training, where an agent interacts with multiple environments asynchronously with a single set of model parameters, is supported for A3C, ACER (Wang et al., 2017), N-step Q-learning, and Path Consistency Learning (PCL). To train an asynchronous agent, one can simply initialize an asynchronous agent and train it using `experiments.train_agent_async`. Batch training refers to synchronous parallel training, where a single agent interacts with multiple environments synchronously in parallel, and is supported for all algorithms for which asynchronous training is not supported. In `ChainerRL`, users can easily perform batch training of agents by initializing an agent and using `experiments.train_agent_batch_with_evaluation`. Many algorithms require additional infrastructure to support recurrent training, e.g., by storing and managing the recurrent state, and managing sequences of observations as opposed to individual observations. `ChainerRL` abstracts these difficulties away from the user, making it simple to employ recurrent architectures for the majority of algorithms. Note that most of the algorithms implemented in `ChainerRL` do not have support for recurrence or batch training in their original published form. In `ChainerRL`, we have added this additional support for most algorithms, as summarized in Table 1.

Algorithm	Discrete Action	Continuous Action	Recurrent Model	Batch Training	CPU Async Training
DQN (Double DQN, SARSA, etc.)	✓	✓(NAF)	✓	✓	✗
Categorical DQN	✓	✗	✓	✓	✗
Rainbow	✓	✗	✓	✓	✗
IQN (and Double IQN)	✓	✗	✓	✓	✗
A3C	✓	✓	✓	✓(A2C)	✓
ACER	✓	✓	✓	✗	✓
NSQ (N-step Q-learning)	✓	✓(NAF)	✓	✗	✓
PCL (Path Consistency Learning)	✓	✓	✓	✗	✓
DDPG	✗	✓	✓	✓	✗
PPO	✓	✓	✓	✓	✗
TRPO	✓	✓	✓	✓	✗
TD3	✗	✓	✗	✓	✗
SAC	✗	✓	✗	✓	✗

Table 1: Summarized list of `ChainerRL` algorithms and their additional supported features.

Appendix B. Reproducibility Results

For each of our reproducibility scripts, we provide the training times of the script (in our repository), full tables of our achieved scores, and comparisons of these scores against those reported in the literature. Though `ChainerRL` has high-quality implementations of dozens of algorithms, we currently have created such "reproducibility scripts" for 9 algorithms. In the Atari benchmark (Bellemare et al., 2013), we have successfully reproduced DQN, IQN, Rainbow, and A3C. For the OpenAI Gym Mujoco benchmark tasks, we have successfully reproduced DDPG, TRPO, PPO, TD3, and SAC.

The reproducibility scripts emphasize correctly reproducing evaluation protocols, which are particularly relevant when evaluating Atari agents. Unfortunately, evaluation protocols tend to vary across papers, and consequently results are often inconsistently reported across the literature (Machado et al., 2018), significantly impacting results. The critical details of standard Atari evaluation protocols are as follows:

Evaluation frequency The frequency (in timesteps) at which the evaluation phase occurs.

Evaluation phase length The number of timesteps in the offline evaluation.

Evaluation episode length The maximum duration of an evaluation episode.

Evaluation policy The policy to follow during an evaluation episode.

Reporting protocol Each intermediate evaluation phase outputs some score, representing the mean score of all evaluation episodes during that evaluation phase. Papers typically report scores according to one of the following reporting protocols:

1. *best-eval*: Papers using the *best-eval* protocol report the highest mean score across all intermediate evaluation phases.
2. *re-eval*: Papers using the *re-eval* protocol report the score of a re-evaluation of the network parameters that produced the *best-eval*.

During a typical Atari agent's 50 million timesteps of training, it is evaluated periodically in an offline evaluation phase for a specified number of timesteps before resuming training. Since most papers report final results using the best model as determined by these periodic evaluation phases, the frequency of evaluation is key, as it provides the author of a paper with more models to select from when reporting final results. The length of the evaluation phase is important, because shorter evaluation phases have higher variance in performance and longer evaluation phases have less variance in performance. Again, since these intermediate evaluations are used in some way when reporting final performance, the length of the evaluation phase is important when reproducing results. The length of the evaluation episodes can impact performance, as permitting the agent to have longer episodes may allow it to accrue more points. Oftentimes, since the agent performs some form of exploratory policy during training, the agent sometimes changes policies specifically for evaluations. Each of the listed details, especially the reporting protocols, can significantly influence the results, and thus are critical details to hold consistent for a fair comparison between algorithms.

Table 2 lists the results obtained by `ChainerRL`'s reproducibility scripts for DQN, IQN, Rainbow, and A3C on the Atari benchmark, with comparisons against a published result. Table 3 depicts the evaluation protocol used for each algorithm, with a citation of the source paper whose results we compare against. Note that the results for the A3C (Mnih et al., 2016) algorithm do not come from the original A3C paper, but from another (Fortunato et al., 2018). For continuous-action algorithms, the results on `OpenAI Gym` MuJoCo tasks for DDPG (Lillicrap et al., 2016), TRPO (Schulman et al., 2015), PPO (Schulman et al., 2017), TD3 (Fujimoto et al., 2018), and SAC (Haarnoja et al., 2018) are reported in Table 4. For all algorithms and environments listed in tables 2 and 4, we have released models trained through our reproducibility scripts, which researchers can use.

The reproducibility scripts are produced through a combination of reading released source code and studying published hyperparameters, implementation details, and evaluation protocols. We also have extensive email correspondences with authors to clarify ambiguities, omitted details, or inconsistencies that may exist in papers.

As seen in both the Atari and MuJoCo reproducibility results, sometimes a reproduction effort cannot be directly compared against the original paper's reported results. For example, the reported scores in the original paper introducing the A3C algorithm (Mnih et al., 2016) utilize demonstrations that are not publicly available, making it impossible to accurately compare a re-implementation's scores to the original paper. In such scenarios, we seek out high-quality published research (Fortunato et al., 2018; Henderson et al., 2018; Fujimoto et al., 2018) from which faithful reproductions are indeed possible, and compare against these.

	DQN		IQN		Rainbow		A3C	
Game	**CRL**	**Published**	**CRL**	**Published**	**CRL**	**Published**	**CRL**	**Published**
Air Raid	6450.5 ± 5.9e+2	-	9933.5 ± 4.9e+2	-	6754.3 ± 2.4e+2	-	3923.8 ± 1.5e+2	-
Alien	1713.1 ± 2.3e+2	**3069**	**12049.2** ± 8.9e+2	7022	**11255.4** ± 1.6e+3	9491.7	2005.4 ± 4.3e+2	**2027**
Amidar	**986.7** ± 1.0e+2	739.5	2602.9 ± 3.9e+2	**2946**	3302.3 ± 7.2e+2	**5131.2**	869.7 ± 7.7e+1	**904**
Assault	3317.2 ± 7.3e+2	**3359**	24315.8 ± 9.6e+2	**29091**	**17040.6** ± 2.0e+3	14198.5	**6832.6** ± 2.e+3	2879
Asterix	5936.7 ± 7.3e+2	**6012**	**484527.4** ± 7.4e+4	342016	**440208.0** ± 9.e+4	428200.3	**9363.0** ± 2.8e+3	6822
Asteroids	1584.5 ± 1.6e+2	**1629**	**3806.2** ± 1.5e+2	2898	**3274.9** ± 8.4e+2	2712.8	**2775.6** ± 3.3e+2	2544
Atlantis	**96456.0** ± 6.5e+3	85641	937491.7 ± 1.6e+4	**978200**	**895215.8** ± 1.3e+4	826659.5	**836040.0** ± 4.7e+4	422700
Bank Heist	**645.0** ± 4.7e+1	429.7	1333.2 ± 2.3e+1	**1416**	**1655.1** ± 1.0e+2	1358.0	**1321.6** ± 6.6e+0	1296
Battle Zone	5313.3 ± 2.9e+3	**26300**	**67834.0** ± 5.1e+3	42244	**87015.0** ± 1.3e+4	62010.0	7998.0 ± 2.6e+3	**16411**
Beam Rider	**7042.9** ± 5.2e+2	6846	40077.2 ± 4.1e+3	**42776**	**26672.1** ± 8.3e+3	16850.2	9044.4 ± 4.7e+2	**9214**
Berzerk	707.2 ± 1.7e+2	-	**92830.5** ± 1.6e+5	1053	**17043.4** ± 1.2e+4	2545.6	**1166.8** ± 3.8e+2	1022
Bowling	**52.3** ± 1.2e+1	42.4	85.8 ± 6.1e+0	**86.5**	**55.7** ± 1.5e+1	30.0	31.3 ± 2.4e-1	**37**
Boxing	**89.6** ± 3.1e+0	71.8	**99.9** ± 2.1e-2	99.8	**99.8** ± 1.3e-1	99.6	**96.0** ± 1.9e+0	91
Breakout	364.9 ± 3.4e+1	**401.2**	665.2 ± 1.1e+1	**734**	353.0 ± 1.1e+1	**417.5**	**569.9** ± 1.9e+1	496
Carnival	5222.0 ± 2.9e+2	-	5478.7 ± 4.6e+2	-	4762.8 ± 6.6e+2	-	4643.3 ± 1.2e+3	-
Centipede	5112.6 ± 6.9e+2	**8309**	10576.6 ± 1.7e+3	**11561**	**8220.1** ± 4.6e+2	8167.3	**5352.4** ± 3.3e+2	5350
Chopper Command	6170.0 ± 1.6e+3	**6687**	**39400.9** ± 7.4e+3	16836	**103942.2** ± 1.7e+5	16654.0	**6997.1** ± 4.5e+3	5285
Crazy Climber	108472.7 ± 1.5e+3	**114103**	178080.2 ± 3.0e+3	**179082**	**174438.8** ± 1.8e+4	168788.5	121146.1 ± 2.6e+3	**134783**
Demon Attack	9044.3 ± 1.8e+3	**9711**	**135497.1** ± 1.5e+3	128580	101076.9 ± 1.1e+4	**111185.2**	**111339.2** ± 6.3e+3	37085
Double Dunk	**-9.7** ± 1.8e+0	-18.1	5.6 ± 1.4e+1	5.6	-1.0 ± 7.9e-1	**-0.3**	1.5 ± 3.5e-1	**3**
Enduro	298.2 ± 5.4e+0	**301.8**	**2363.6** ± 3.3e+0	2359	**2278.6** ± 4.1e+0	2125.9	0.0 ± 0.e+0	0
Fishing Derby	**11.6** ± 7.6e+0	-0.8	**38.8** ± 4.3e+0	33.8	**44.6** ± 5.1e+0	31.3	**38.7** ± 1.6e+0	-7
Freeway	8.1 ± 1.3e+1	**30.3**	34.0 ± 0.e+0	34.0	33.6 ± 4.6e-1	**34.0**	0.0 ± 7.3e-3	0
Frostbite	**1093.9** ± 5.5e+2	328.3	**8196.1** ± 1.5e+3	4342	**10071.6** ± 8.6e+2	9590.5	**288.2** ± 2.9e+1	288
Gopher	8370.0 ± 1.1e+3	**8520**	117115.0 ± 2.8e+3	**118365**	**82497.8** ± 5.6e+3	70354.6	**9251.0** ± 1.8e+3	7992
Gravitar	**445.7** ± 5.e+1	306.7	**1006.7** ± 2.5e+1	911	**1605.6** ± 1.9e+2	1419.3	244.5 ± 4.4e+0	**379**
Hero	**20538.7** ± 2.0e+3	19950	**28429.4** ± 2.4e+3	28386	27830.8 ± 1.3e+4	**55887.4**	**36599.2** ± 3.5e+2	30791
Ice Hockey	-2.4 ± 4.3e-1	**-1.6**	0.1 ± 2.0e+0	**0.2**	**5.7** ± 5.4e-1	1.1	-4.5 ± 1.9e-1	**-2**
Jamesbond	**851.7** ± 2.3e+2	576.7	26033.6 ± 3.8e+3	**35108**	24997.6 ± 5.6e+3	-	376.9 ± 2.6e+1	**509**
Journey Escape	-1894.0 ± 5.8e+2	-	-632.9 ± 9.7e+1	-	-429.2 ± 4.4e+2	-	-989.2 ± 4.2e+1	-
Kangaroo	**8831.3** ± 6.8e+2	6740	**15876.3** ± 6.4e+2	15487	11038.8 ± 5.8e+3	**14637.5**	252.0 ± 1.2e+2	**1166**
Krull	**6215.0** ± 2.3e+3	3805	9741.8 ± 1.2e+2	**10707**	8237.9 ± 2.2e+2	**8741.5**	8949.3 ± 8.5e+2	**9422**
Kung Fu Master	**27616.7** ± 1.3e+3	23270	**87648.3** ± 1.1e+4	73512	33628.2 ± 9.5e+3	**52181.0**	**39676.3** ± 2.4e+3	37422
Montezuma Revenge	0.0 ± 0.e+0	0.0	**0.4** ± 6.8e-1	0.0	16.2 ± 2.2e+1	**384.0**	2.8 ± 6.3e-1	**14**
Ms Pacman	**2526.6** ± 1.e+2	2311	5559.7 ± 4.5e+2	**6349**	**5780.6** ± 4.6e+2	5380.4	**2552.9** ± 1.9e+2	2436
Name This Game	7046.5 ± 2.0e+2	**7257**	**23037.2** ± 2.e+2	22682	**14236.4** ± 8.5e+2	13136.0	**8646.0** ± 3.e+3	7168
Phoenix	7054.4 ± 1.9e+3	-	**125757.5** ± 3.6e+4	56599	84659.6 ± 1.4e+5	**108528.6**	**38428.3** ± 3.1e+3	9476
Pitfall	-28.3 ± 2.1e+1	-	0.0 ± 0.e+0	0.0	-3.2 ± 2.9e+0	**0.0**	-4.4 ± 2.9e+0	**0**
Pong	**20.1** ± 4.0e-1	18.9	21.0 ± 0.e+0	21.0	**21.0** ± 6.4e-2	20.9	**20.7** ± 3.9e-1	7
Pooyan	3118.7 ± 3.5e+2	-	27222.4 ± 9.9e+3	-	7772.7 ± 3.6e+2	-	4237.9 ± 5.8e+1	-
Private Eye	1538.3 ± 1.3e+3	**1788**	**259.9** ± 1.0e+2	200	99.3 ± 5.8e-1	**4234.0**	449.0 ± 1.6e+2	**3781**
Qbert	10516.0 ± 2.6e+3	**10596**	25156.8 ± 5.3e+2	**25750**	**41819.6** ± 1.9e+3	33817.5	**18889.2** ± 7.6e+2	18586
Riverraid	7784.1 ± 6.8e+2	**8316**	**21159.7** ± 8.0e+2	17765	26574.2 ± 1.8e+3	-	12683.5 ± 5.3e+2	-
Road Runner	**37092.0** ± 3.e+3	18257	**65571.3** ± 5.6e+3	57900	**65579.3** ± 6.1e+3	62041.0	40660.6 ± 2.1e+3	**45315**
Robotank	47.4 ± 3.6e+0	**51.6**	**77.0** ± 1.3e+0	62.5	**75.6** ± 2.1e+0	61.4	3.1 ± 5.1e-2	**6**
Seaquest	**6075.7** ± 2.3e+2	5286	26042.3 ± 3.9e+3	**30140**	3708.5 ± 1.7e+3	**15898.9**	**1785.6** ± 4.1e+0	1744
Skiing	-13030.2 ± 1.2e+3	-	-9333.6 ± 7.4e+1	**-9289**	**-10270.9** ± 8.6e+2	-12957.8	-13094.2 ± 3.7e+3	**-12972**
Solaris	1565.1 ± 6.e+2	-	7641.6 ± 8.2e+2	**8007**	**8113.0** ± 1.2e+3	3560.3	3784.2 ± 3.5e+2	**12380**
Space Invaders	1583.2 ± 1.5e+2	**1976**	**36952.7** ± 2.9e+4	28888	17902.6 ± 1.3e+4	**18789.0**	**1568.9** ± 3.7e+2	1034
Star Gunner	56685.3 ± 1.0e+3	**57997**	**182105.3** ± 1.9e+4	74677	**188384.2** ± 2.3e+4	127029.0	**60348.7** ± 2.6e+3	49156
Tennis	-5.4 ± 7.6e+0	**-2.5**	**23.7** ± 1.7e-1	23.6	-0.0 ± 2.4e-2	0.0	-12.2 ± 4.3e+0	**-6**
Time Pilot	5738.7 ± 9.0e+2	**5947**	**13173.7** ± 7.4e+2	12236	**24385.2** ± 3.5e+3	12926.0	4506.6 ± 2.8e+2	**10294**
Tutankham	141.9 ± 5.1e+1	**186.7**	**342.1** ± 8.2e+0	293	**243.2** ± 2.9e+1	241.0	**296.7** ± 1.8e+1	213
Up N Down	**11821.5** ± 1.1e+3	8456	73997.8 ± 1.7e+4	**88148**	291785.9 ± 7.3e+3	-	**95014.6** ± 5.1e+4	89067
Venture	**656.7** ± 5.5e+2	380.0	656.2 ± 6.4e+2	**1318**	**1462.3** ± 3.4e+1	5.5	0.0 ± 0.e+0	0
Video Pinball	9194.5 ± 6.3e+3	**42684**	664174.2 ± 1.1e+4	**698045**	477238.7 ± 2.6e+4	**533936.5**	**377939.3** ± 1.8e+5	229402
Wizard Of Wor	1957.3 ± 2.7e+2	**3393**	23369.5 ± 5.4e+3	**31190**	**20695.0** ± 9.e+2	17862.5	2518.7 ± 5.1e+2	**8953**
Yars Revenge	4397.3 ± 2.1e+3	-	**30510.0** ± 2.3e+2	28379	86609.9 ± 1.e+4	**102557.0**	19663.9 ± 6.6e+3	**21596**
Zaxxon	**5698.7** ± 1.0e+3	4977	16668.5 ± 3.4e+3	**21772**	**24107.5** ± 2.4e+3	22209.5	78.9 ± 6.8e+0	**16544**
# **Higher scores**	22	26	28	23	34	17	27	24
# **Ties**	1		4		1		3	
# **Seeds**	5	1	3	1	3	1	5	3

Table 2: The performance of `ChainerRL` (± standard deviation) against published results on Atari benchmarks.

	DQN	IQN	Rainbow	A3C
Eval Frequency (timesteps)	250K	250K	250K	250K
Eval Phase (timesteps)	125K	125K	125K	125K
Eval Episode Length (time)	5 min	30 min	30 min	30 min
Eval Episode Policy	$\epsilon = 0.05$	$\epsilon = 0.001$	$\epsilon = 0.0$	N/A
Reporting Protocol	*re-eval*	*best-eval*	*re-eval*	*best-eval*

Table 3: Evaluation protocols used for the Atari reproductions. The evaluation protocols of DQN, IQN, Rainbow, and A3C match the evaluation protocols used by Mnih et al. (2015), Dabney et al. (2018), Hessel et al. (2017), and Fortunato et al. (2018), respectively. An evaluation episode policy with an ϵ indicates that the agent performs an ϵ-greedy evaluation.

	DDPG (Fujimoto et al., 2018)		**TD3** (Fujimoto et al., 2018)	
Environment	**CRL**	**Published**	**CRL**	**Published**
HalfCheetah-v2	**10325.45**	8577.29	**10248.51** ± 1063.48	9636.95 ± 859.065
Hopper-v2	**3565.60**	1860.02	**3662.85** ± 144.98	3564.07 ± 114.74
Walker2d-v2	**3594.26**	3098.11	**4978.32** ± 517.44	4682.82 ± 539.64
Ant-v2	**774.46**	888.77	**4626.25** ± 1020.70	4372.44 ± 1000.33
Reacher-v2	**-2.92**	-4.01	**-2.55** ± 0.19	-3.60 ± 0.56
InvertedPendulum-v2	902.25	**1000.00**	1000.00 ± 0.0	1000.00 ± 0.0
InvertedDoublePendulum-v2	7495.56	**8369.95**	8435.33 ± 2771.39	**9337.47** ± 14.96

	TRPO (Henderson et al., 2018)		**PPO** (Henderson et al., 2018)		**SAC** (Haarnoja et al., 2018)	
Environment	**CRL**	**Published**	**CRL**	**Published**	**CRL**	**Published**
HalfCheetah-v2	**1474** ± 112	205 ± 256	**2404** ± 185	2201 ± 323	14850.54	~15000
Hopper-v2	**3056** ± 44	2828 ± 70	2719 ± 67	**2790** ± 62	2911.89	~3300
Walker2d-v2	3073 ± 59	-	2994 ± 113	-	5282.61	~5600
Ant-v2	-	-	-	-	5925.63	~5800
Swimmer-v2	200 ± 25	-	111 ± 4	-	-	-
Humanoid-v2	-	-	-	-	7772.08	~8000

Table 4: The performance of `ChainerRL` against published baselines on `OpenAI Gym` MuJoCo benchmarks. For DDPG and TD3, each `ChainerRL` score represents the maximum evaluation score during 1M-step training, averaged over 10 trials with different random seeds, where each evaluation phase of ten episodes is run after every 5000 steps. For PPO and TRPO, each `ChainerRL` score represents the final evaluation of 100 episodes after 2M-step training, averaged over 10 trials with different random seeds. For SAC, each `ChainerRL` score reports the final evaluation of 10 episodes after training for 1M (Hopper-v2), 3M (HalfCheetah-v2, Walker2d-v2, and Ant-v2), or 10M (Humanoid-v2) steps, averaged over 10 trials with different random seeds. Since the original paper (Haarnoja et al., 2018) provides learning curves only, the published scores are approximated visually from the learning curve. The sources of the published scores are cited with each algorithm. We use the v2 environments, whereas some published papers evaluate on the now-deprecated v1 environments.

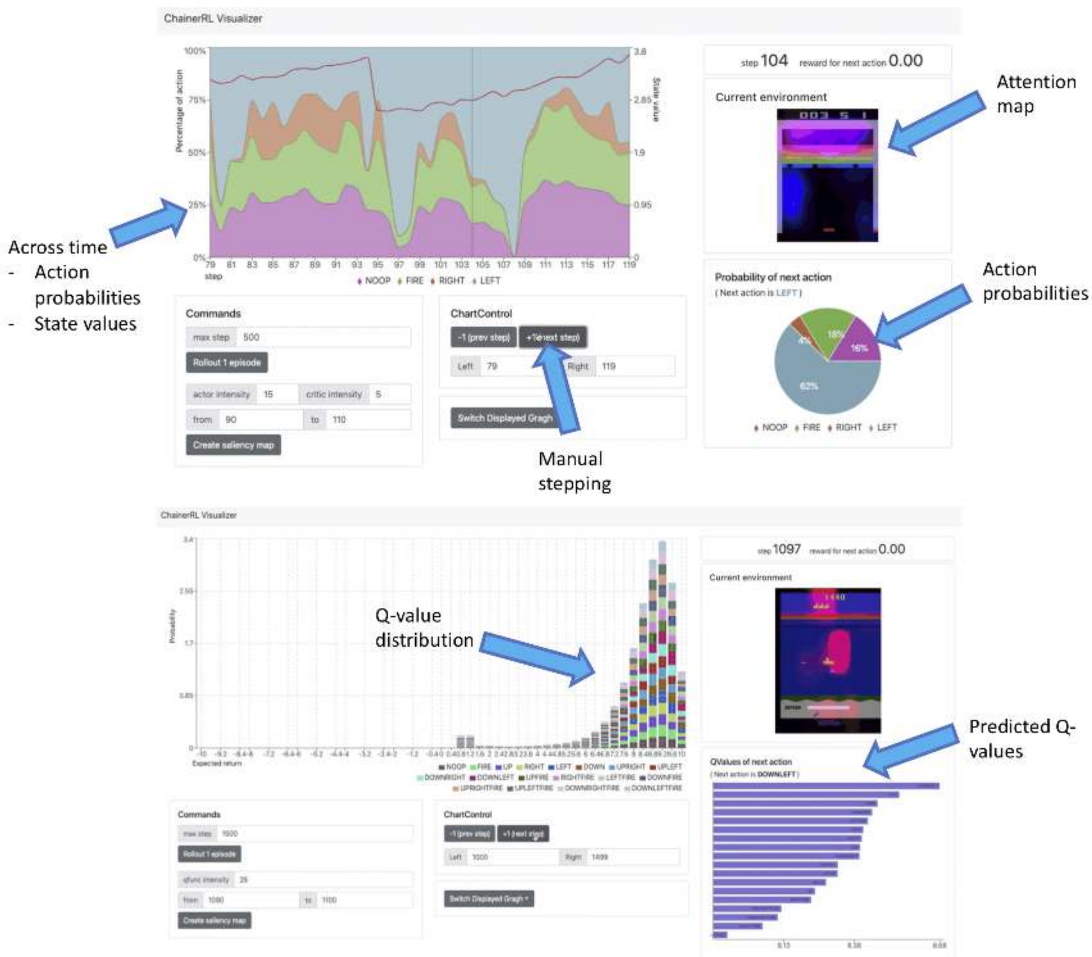

Figure 2: The `ChainerRL Visualizer`. With the `ChainerRL Visualizer`, users can closely investigate an agent's behaviors within a browser window. *top*: Visualization of a trained A3C agent on Breakout. *bottom*: Visualization of a C51 (Bellemare et al., 2017) agent trained on Seaquest.

Appendix C. Visualizer Images

Figure 2 depicts some of the key features of the `ChainerRL Visualizer` for an actor-critic algorithm and a distributional value-based algorithm. The top of the figure depicts a trained A3C agent in the Atari game Breakout. With the visualizer, one can visualize the portions of the pixel input that the agent is attending to as a saliency map (Greydanus et al., 2018). Additionally, users can perform careful, controlled investigations of agents by manually stepping through an episode, or can alternatively view rollouts of agents. Since A3C is an actor-critic agent with a value function and a policy outputting a distribution over actions, we can view the probabilities with which the agent will perform a specific action, as well as the agent's predicted state values. If the agent learns Q-values or a distribution of Q-values, the predicted Q-value or Q-value distribution for each action can be displayed, as shown in the bottom of Figure 2.

Appendix D. Pseudocode

The set of algorithms that can be developed by combining the agent building blocks of `ChainerRL` is large. One notable example is Rainbow (Hessel et al., 2017), which combines double updating (Van Hasselt et al., 2016), prioritized replay (Schaul et al., 2016), N-step learning, dueling architectures (Wang et al., 2016), and Categorical DQN (Bellemare et al., 2017) into a single agent. The following pseudocode depicts the simplicity of creating and training a Rainbow agent with `ChainerRL`.

```python
import chainerrl as crl
import gym

q_func = crl.q_functions.DistributionalDuelingDQN(...)# dueling
crl.links.to_factorized_noisy(q_func) # noisy networks
# Prioritized Experience Replay Buffer with a 3-step reward
per = crl.replay_buffers.PrioritizedReplayBuffer(num_step_return=3,...)
# Create a rainbow agent
rainbow = crl.agents.CategoricalDoubleDQN(per, q_func,...)
num_envs = 5 # Train in five environments
env = crl.envs.MultiprocessVectorEnv(
    [gym.make("Breakout") for _ in range(num_envs)])

# Train the agent and collect evaluation statistics
crl.experiments.train_agent_batch_with_evaluation(rainbow, env, steps=...)
```

We first create a distributional dueling Q-function, and then in a single line, convert it to a noisy network. We then initialize a prioritized replay buffer configured to use N-step rewards. We pass this replay buffer to `ChainerRL`'s built-in `CategoricalDoubleDQN` agent to produce a Rainbow agent. Moreover, with `ChainerRL`, users can easily specify the number of environments in which to train the Rainbow agent in synchronous parallel processes, and the `experiments` module will automatically manage the training loops, evaluation statistics, logging, and saving of the agent.